Secure Session Protocol - Concept and Implementation of a Protocol to Securely Operate Web Applications

Secure Session Protocol - Konzept und Implementierung eines Protokolls zum sicheren Betrieb von Web Anwendungen

Master-Thesis von Florian Oswald aus Weinheim

August 2014

1. Prüfer: Prof. Dr. Michael Waidner
2. Prüfer: Marco Ghiglieri

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Fachgebiet Sicherheit in der Informationstechnik

CASED

EC SPRIDE

Secure Session Protocol - Concept and Implementation of a Protocol to Securely Operate Web Applications
Secure Session Protocol - Konzept und Implementierung eines Protokolls zum sicheren Betrieb von Web Anwendungen

Vorgelegte Master-Thesis von Florian Oswald aus Weinheim

1. Prüfer: Prof. Dr. Michael Waidner
2. Prüfer: Marco Ghiglieri

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 4.August 2014

(Florian Oswald)

Abstract

The importance of web services has steadily increased over the last couple of years. Furthermore, web services have often the requirement to be usable on any device, including standalone computer as well as mobile devices like smartphones and tablets. With the development of advanced web technologies, even more applications are realized as web services. This also includes web services that process sensitive data.

Therefore, these web services are a profitable target for adversaries to steal sensitive data of users using the web service. The used attacks are become more complex and often using a combination of several security vulnerabilities to successfully break into the system. On the other hand, new web security techniques are introduced to mitigate upcoming attacks by adversaries. However, a proper client-side security is not deployed as of today.

This work **Secure Session Protocol - Concept and Implementation of a Protocol to Securely Operate Web Applications**, describes an advanced security concept, which provides web services a mechanism to increase the client-side security and therefore the overall security of the used web service. The presented concept discusses the *Secure Session Protocol*, which enables the web service to enforce security properties on the client-side. Together with a trustworthy partner of the web service on the client-side, the so called *Secure Session Protocol Extension*(SSP-Extension), the web service creates a *Secure Session* at the client-browser. The security properties of the Secure Session are based on correct information collected by the SSP-Extension beforehand. In order to securely exchange Secure Session Protocol related information, the web service and the SSP-Extension are using a covert channel. This channel is secured through the *Secure Session Key*. Together with the HTTPS certificate of the web service, the protocol provides mutual identification. Finally, we show an example implementation of the Secure Session Protocol.

Kurzfassung

Aktuelle Web Browser und damit auch Web Anwendungen gewinnen immer mehr an Bedeutung. Die geforderte Nutzbarkeit aller Anwendungen auf mobilen und stationären Endgeräten führt dazu, dass immer mehr Anwendungsfälle vom klassischen Programm in das Internet wandern. Die Komplexität der Anwendungen, die abgebildet werden muss, steigt stetig. Zusätzlich dazu erhöht sich auch der Anteil an Web Anwendungen, welcher sensitive Daten verarbeitet. Als Folge daraus steigt das Interesse eines Angreifers Schwachstellen einer Anwendung auszunutzen, um an sensitive Daten zu gelangen. Man kann beobachten, dass einige Schwachstellen sehr lange bekannt sind und andere durch sehr komplexe Konstellationen erst geöffnet werden. Aktuell gibt es für die meisten Probleme eine Lösung, allerdings sind diese oft nicht effektiv eingesetzt um alle Gefahren sicher abwehren zu können. Sicherheitsgarantien für den Betreiber einer Web Anwendung gibt es im Allgemeinen nicht.

Die Arbeit „Secure Session Protocol - Konzept und Implementierung eines Protokolls zum sicheren Betrieb von Web Anwendungen“ stellt ein erweitertes Sicherheitskonzept für Web Anwendungen vor, welches den Browser gegen clientseitige Angriffe besser schützen soll und der Web Anwendungen die Möglichkeit gibt, die Umgebung auf der Clientseite besser zu kontrollieren. Dem Provider der Web Anwendung wird durch das „Secure Session Protocol“ ein Mechanismus vorgestellt, der es ermöglicht, Web Sicherheitskonzepte innerhalb des Client-Browsers sichergestellt umzusetzen. Dazu wird eine sogenannte „Secure Session“ zwischen der Web Anwendung und dem Browser des Nutzers aufgebaut, welche alle Sicherheitsanforderungen des Providers der Web Anwendung erfüllt. Die Sicherheitsanforderungen werden anhand von authentischen Informationen über den Client-Browser erstellt. Dazu wird eine Erweiterung innerhalb des Client-Browsers verwendet, welche als vertrauenswürdiger Partner der Web Anwendung auf der Seite des Nutzers agiert und dafür sorgt, dass alle Regeln der „Secure Session“ während der Nutzung der Web Anwendung aktiv sind. Dazu verwendet die Erweiterung und die Web Anwendung einen separaten verschlüsselten Kanal, welcher durch den „Secure Session Key“ abgesichert wird. Zusammen mit dem HTTPS Protokoll wird dadurch zusätzlich eine beidseitige Identifizierung ermöglicht. Die folgende Arbeit stellt das gesamte Konzept des „Secure Session Protocols“ vor und zeigt anhand einer beispielhaften Implementierung grundlegende Sicherheitskonzepte des Protokolls.

Contents

1 Introduction

The number of internet users has steadily increased since the beginning of the internet. Looking fifteen years back, in 1999 a total of 413 million users were using the internet, as in contrast today, more than 40 percent of the world population (2.756.000 of 7.138.000 people) are using the internet [58].

This development is based on the fact, that the internet can be used nearly everywhere today. Even more devices are equipped with an internet connection (called connected devices) and therefore can be used to access the internet. In 2014, more than 1.5 billion connected devices were sold [55]. Besides the computer and laptop, smartphones and tablets become more popular for using the internet. Also the industry has identified this trend and has ported the internet onto several other electronics devices. For example, current TVs are equipped with an internet connection. This gives users the opportunity to browse the internet while watching TV. These new television devices are called Smart TVs. In figure 1 the current number of connected devices and the further development is visualized.

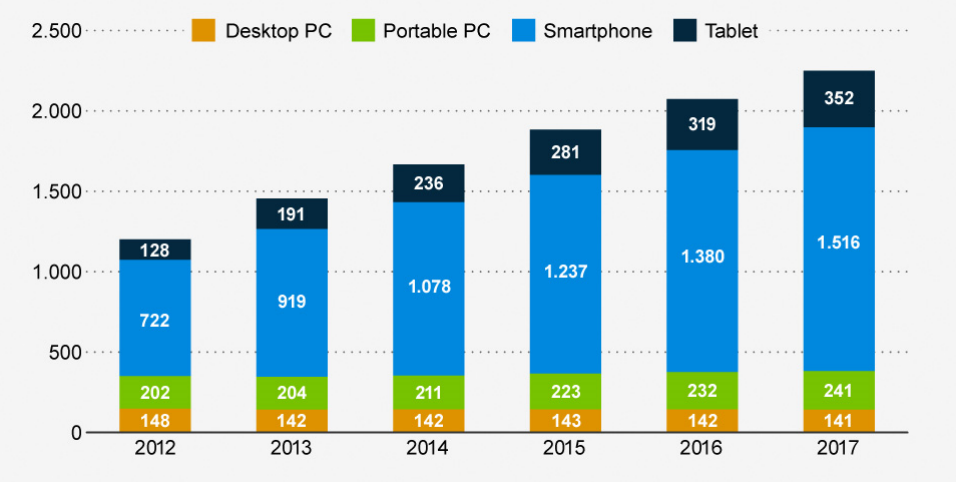

Figure 1: Usage of connected devices (in million) - Copyright Statista 2014

Further, a **mobile internet connection** gives users the possibility to use the internet everywhere by using cellular internet connections or wireless LAN hotspots. For example, in 2013, 51 percent of (German) internet users were using mobile connections to access the internet [56].

The network infrastructure has been further developed in the last years, creating space for new forms of services like cloud based web services. Also, new innovative web technologies like HTML5 and CSS3 were introduced in the last years, providing new tools for developers. Hence, today it is possible to create even more advanced web services for the user. These new web services provide better usability and guarantee access to the web content from everywhere and thus building a profitable environment for companies. Therefore, more and more use cases are implemented as a web service instead of a standalone computer version. These web services are easy to use and accessible with any device.

Furthermore, uses cases operating with sensitive data of the user are more common. The term sensitive data describes *personally identifiable information* [29], which for example is related to health or financial activities. As an example, the reader might consider a web service which enables the user to manage his banking account online. This includes operations like checking the current balance of the banking account or securely executing banking transactions. In 2013, 45 percent of the German population (in total 35 million people) with a banking account were using online banking web services to manage their banking related transactions [57]. Further, several banks have reacted to this trend and are solely offering online banking accounts [10].

A web service operating with sensitive data has additionally the requirement to be secure against any kind of web security attack. Therefore, web services need to guarantee the security goals *confidentiality, privacy, authenticity and integrity* and therefore *nonrepudiability*. To achieve these goals, new web security mechanism are developed and introduced throughout the last years to further secure web services. One recently published web security mechanism is the *Content Security Policy* [1] developed by the Mozilla Foundation [63]. The goal of the Content Security Policy is to mitigate popular web security attacks like XSS, Clicking and Packet Sniffing [40].

Despite the fact that new advanced web security mechanisms are introduced, web security attacks are still present. Looking at the example of online banking web services, the online banking accounts of internet users have become one of the main targets of adversaries. In 2013, the Federal Criminal Police (BKA) in Germany [5] has reported 4.100 incidents

of adversary attacks against online banking accounts. The number of reported accidents therefore has increased by over 20 percent in comparison to 2012 [12]. Further, in 2014, 17 percent of the online banking users have reported that in their social environment people were financially damaged through adversary attacks on their online banking account [59].

The question is, why is it not possible for a web service to guarantee a well-defined security level for the user on the client-side? Theoretically, as by using all available web security mechanism most web security related attacks can be omitted?

Current web services have no possibility to **verify** that certain web security rules are enforced on the client-side. For example, an adversary controlling the client-browser (so called Man-In-The-Browser) can easily disable the security rules before they are even implemented. Therefore, the web service currently has to trust the client-browser, that all sent web security rules are executed and enforced.

This work addresses the problem of the enforcement and verification of web security rules on the client-side. The concept of the Secure Session Protocol is introduced, which gives the web service the possibility to enforce web security rules on the client-side. Further, the Secure Session Protocol provides a mechanism for client-side identification and enables the web service to receive correct information about the requesting client-browser to individually create client-side specific security rules.

The following work starts with a classification of different web services. The work continues with an overview and description of possible web security attacks that an adversary is able to execute. Before starting with the explanation of the Secure Session Protocol, the work continuous with an overview of related web security concepts. The next chapter, shortly summarizes the problem from the perspective of the web service and further underlines the need for an advanced web security concept. The main part of this work starts with the description a general description of the Secure Session Protocol. Chapter 3 includes a description of the different components, participants and key concepts of the protocol. The main chapter then continues with a detailed explanation of the single protocol states. The chapter concludes with an example of the Secure Session Protocol. The achieved Security Model is explained in chapter 4. In a section of chapter 4, the security mechanism are evaluated against the OWASP Top 10, a listing of the Top 10 web security risk. To underline the concept of the Secure Session Protocol, an explanation of a concept implementation of the Secure Session Protocol is given in chapter 6. The work concludes with an outlook at currently open points in section 6.4, regarding the development and implementation of the Secure Session Protocol and concludes with an Outlook.

1.1 Classification of Web Services

In order to get a better understanding for which kind of web service the Secure Session Protocol is developed, we classify web service into three different groups. Relevant for this classification is their usage of sensitive data. Nevertheless, the Secure Session Protocol can be used by every web service that implements the appropriate protocol behavior. In figure 2 an overview of the standardization of web services (Type I - Type III) is shown. To classify a web service, the used transfer protocol (HTTP [43] and HTTPS [44]) is examined. In the following section, each class is described and an example for a web service is given.

Note: *By definition, web services handling sensitive data are serving their data solely via HTTPS. However, the Secure Session Protocol is also able to work with HTTP only and therefore no HTTPS encryption is required.*

A **Type I** web service handles no sensitive data at all. The used protocol for transferring the content of the web service over the network is the standard unencrypted HTTP protocol [43], throughout the entire web service. Attacking a Type I web service has no added value for an adversary in terms of retrieving sensitive data from the user. Therefore, the web service has no extra security measurements. An example for a Type I web service is any web page using HTTP only and processing no sensitive data, e.g., a simple online newspaper web service.

The second type, classified as **Type II**, processes sensitive data, but only in a secure environment, which is strictly differentiated from the rest of the web service. The secured environment of the web service is called *private section*. Only in this part of the web service, sensitive data of the user is processed. Thus, the private section is additionally secured by an authentication mechanism, e.g., user name and password. If the user is authenticated against the web service he is clearly identified [1]. The used protocol for transferring sensitive data is the HTTPS protocol, which enables the communication partners to encrypt the whole data traffic. Beside the private section of the web service, there is also a

[1] only in the absence of an adversary

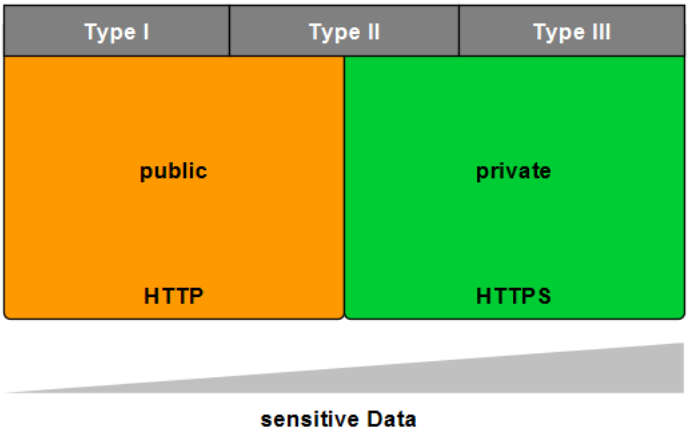

Figure 2: Classification of the web services Type I - Type III

public section that is reachable by any user with no security measurements. The public section of a Type II web service is a Type I web service with all of the properties explained above. As an example for a Type II web service, the reader might consider an online web shop, which has a public section for browsing all available articles and a private section including the shopping basket, payment history and banking information that need to be additionally secured.

The last web service type is called **Type III**. These web services are specialized for processing sensitive data and only have a private section. To enter the private section of a Type III web service, similar to the Type II web services, additional credentials of the user are needed. These credentials are used to authenticate the user to the web service. Besides the HTTPS encryption, additional web security mechanism are used to further secure the web session of the user. An example for a Type III web service is an *online banking service*.

The classification should be a guideline for the reader, to understand for which kind of web services the Secure Session Protocol is suitable. As expected, all web services processing sensitive data should consider using the Secure Session Protocol. Therefore, all Type II and Type III web services should consider implementing the Secure Session Protocol. However, the Secure Session Protocol can be used by any web service without restrictions.

1.2 Adversary Model

This section explains the adversary model, which is used for the remainder of this work. To explain the different web security attacks, we define two adversaries types. Both are visualized in figure 3 to give an overview. Further, this section includes a paragraph about general assumptions that are made to exactly define the security context of the later Secure Session Protocol.

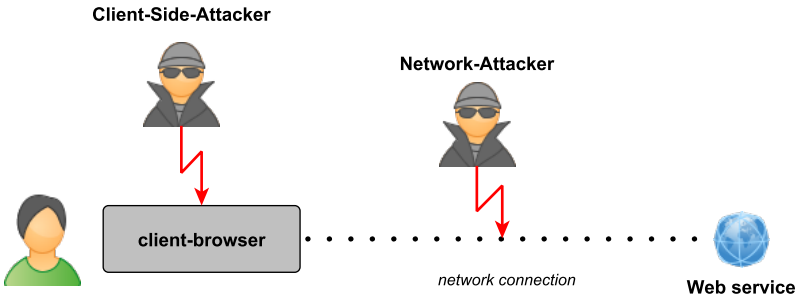

Figure 3: Location of the different attacker types

The first type of an adversary is located at the client-side. In this work, we call this adversary type **Client-Side-Attacker**. The attacker can be active, by attacking the client-browser's integrity or passive by eavesdropping the web session of the

user. Further, the reader might consider a combination of an active and passive adversary at the client-browser. In order to successfully execute the attack, the Client-Side-Attacker needs to exploit vulnerability on the client-side. There are different reasons, why a client-browser is vulnerable to a Client-Side-Attacker, e.g., outdated software components or malicious installed software. In the example, explained in subsection 1.2.1, the adversary is hidden within a custom browser extension. The adversary extension is able to interact with the client-browser and extract data from the current web session of the user. With additional permissions granted to the custom browser extension, the adversary is able to expose sensitive data. Formally, this type of attacker is called **Man-In-The-Browser** [9].

The second type of attacks can be executed by an adversary located between both communication partners. This adversary type is called **Network-Attacker**. We consider the adversary to be an active attacker. Therefore, the attacker adds, deletes and modifies web packets on the network path. In addition, the Network-Attacker can initiate own connections with the communication partners, including the web service and the client-browser. The adversary is a complete Man-In-The-Middle [64]. As for the Client-Side-Attacker, subsection 1.2.1 gives an example for a successful attack of a **Network-Attacker** by executing a Man-In-The-Middle Attack.

Note: *In contrast to the Client-Side-Attacker, we consider the Network-Attacker in this work to be active. Passive attacks, e.g, eavesdropping the connection between the client-browser and the web service, cannot be detected by the Secure Session Protocol presented in this work.*

Assumption

Before continuing with the examples of different web security attacks, we complete the adversary model by defining pre-conditions of the environment. One of the strengths of the Secure Session Protocol depends on the security of a covert channel between the web service and the Secure Session Protocol Extension (as seen in section 4). This additional channel is encrypted with a symmetric encryption key. Therefore, we assume that current **standard cryptographic algorithms**, if used properly, are safe against most adversaries and cannot be broken with currently available methods. Furthermore, the Secure Session Protocol implies that the visited **web service is trustworthy**. This fact is only true for the desired web service by the user. Malicious web services can be identified during the pairing phase of the Secure Session Protocol (see section 3.3.2). The pairing phase is needed (to exchange the symmetric Secure Session Key) before the Secure Session can be established. The pairing depends on an Out-Of-Band [68] exchanged secret. Similar to the cryptographic algorithms, this work assumes that the Out-Of-Band channel is safe against any kind of attacks. In addition, the web service is considered to act properly and safe. The last assumption of the Secure Session Protocol is that the base installation of **the client-browser is valid** and not modified by an attacker before the installation. Modifications of the integrity of the browser afterwards, e.g., through the installation of custom browser extensions or virus software on the computer is considered by the Secure Session Protocol as seen in section 4.

The focus of the Secure Session Protocol is to secure the client-browser against web security attacks on the client-side. The Secure Session Protocol is not designed to mitigate any kind of attacks against the web server and thus the web service. If the web server is compromised a secure execution of Secure Session Protocol is not possible.

Since all types of attackers are described and the assumption for the Secure Session Protocol are made, the section continues with a description of possible attacks executed by the described attacker types. For each of the shown adversary types, one example attack is explained in detail.

1.2.1 Existing Attacks

To further understand the need for an advanced web security concept, this subsection presents possible web security attacks on the client-side. Representative for the different adversary models explained in the previous section, one web security attack example for each attacker type is given. The section starts with an example for a **Client-Side-Attacker**. With the help of a malicious custom browser extension, the adversary is able to *expose sensitive data* of the user on the client-side. Then, an example for a **Network-Attacker** is shown. The explanation of the Network-Attacker example covers the possibility for an attacker to execute SSL-Stripping attacks [39]. As this section only serves for motivation purposes, a complete coverage of all web security attacks is left out in this subsection and can be found in chapter 4.

Client-Side-Attacker

Note: *The used terms for this example are related to Google Chrome's [22] custom browser extensions platform API [24].*

Custom browser extensions can be programmed and distributed by everyone. They are constantly gaining reputation as described in a blog entry by the Mozilla Foundation [62]. According to the blog entry, every third user uses custom

browser extension to modify and customize their own browser. Nevertheless, custom browser extension can be a potential security issue within the client-browser as described in [38].

Before installing a custom browser extension, the user is asked to grant the requested permission to the extension. Described in a report by the security company TrendMicro [65], most (of mobile android) applications request more permissions than they actual need. This problem can be transferred in the context of custom browser extensions (the security concept of extensions is equal to that of android applications). Therefore, custom browser extension from *possibly untrusted sources* are a security and privacy flaw.

The example for a **Client-Side-Attacker** uses a custom browser extension, which has two separate functions. The main purpose of this extension is to show the current IP address of the web service. The extension needs the permission *webRequest* [20] for all requested hosts *(*//*.*/*)*. In the background, hidden from the user, the adversary executes his attack. Using the same permission as the main purpose of the extension, a script executes in the background unrecognized by the user. This script filters sensitive data of the user, by monitoring incoming and outgoing web packets. The collected data is transferred to a server controlled by an attacker via the network connection of the client-browser.

Network-Attacker

The HTTPS web protocol is a secure extension to the standard web protocol HTTP. HTTPS provides secure End-to-End cryptography. The HTTPS protocol additionally uses a certificate system, called Public-Key-Infrastructure (PKI), which enables the user to identify the requesting web service one the base of trust. The data traffic between both communication partners is then securely encrypted. However, for an adversary it is possible to compromise such connections by executing a Man-in-the-Middle attack, which is called SSL-Stripping attack [39].

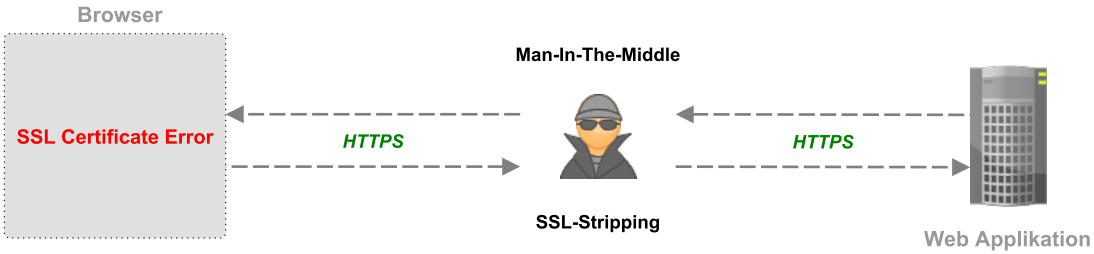

Figure 4: Man-In-The-Middle attacker executes a SSL-Stripping attack

The Man-In-The-Middle attacker is placed between the web service and the client-browser. Therefore, the attacker can only monitor the encrypted traffic. As stated in the previous section, we assume that the adversary is not able to break the used cryptography. To bypass the encryption, the Network-Attacker uses the SSL-Stripping attack. To execute the attack, the adversary establishes an HTTPS connection in both directions (a connection between the client-browser and the adversary and another between the adversary and the web service). The setting of this attacks is visualized in figure 4. If web packets are transmitted, the adversary in the middle decrypts incoming packets, logging the packets and encrypts the packet before sending it to the opposite communication partner.

The SSL-Stripping attack can only be detected by the client-browser, as the detection is based on the certificate system used by the HTTPS connection. The Man-In-The-Middle attacker does not have the original HTTPS certificate of the requested web service. Hence, the delivered HTTPS certificate differs from the original web service one. The attackers uses his own certificate to secure the connection, therefore the used HTTPS certificate might not be valid. Thus, the user is displayed a certificate error within the browser, as shown in figure 5.

The adversary is now able to fully control the web traffic between the web service and the client-browser. Only the client-browser is able to mitigate a SSL-Stripping attack by reacting properly to the certificate error by closing the connection. In the paper [33] the effectiveness of displayed certificate errors of different browser versions is evaluated. As a result of this work, the authors concluding that current certificate error pages are not as useful as needed. Therefore, we assume that SSL Stripping attacks are used in the wild.

The two presented examples only give an impression of possible attacks on the client-side. A ranking of the most used web security attacks is yearly published by the Open Web Application Security Project (OWASP)[46] and is called the

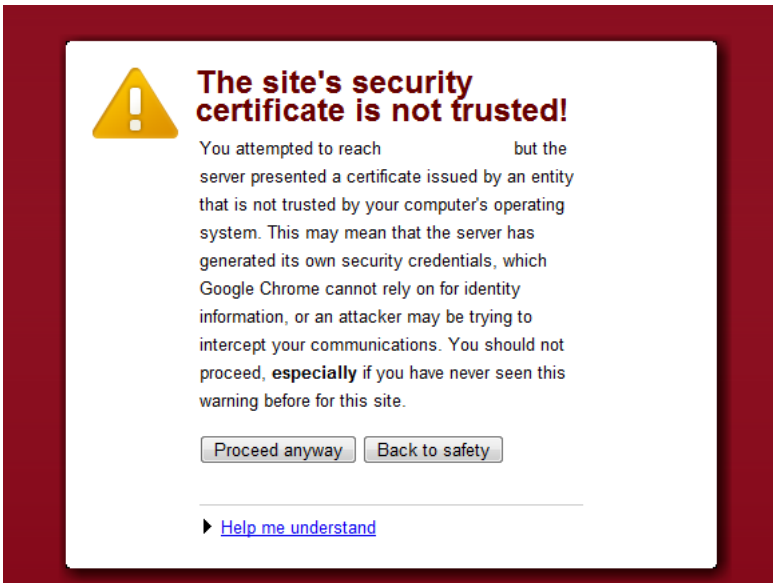

Figure 5: Certificate Error Message from Google Chrome

OWASP Top 10 [45]. In section 4 the established security environment, which is achieved through the Secure Session Protocol is compared with the top web security risk listed in the *OWASP Top 10*.

1.3 Motivation

Explained in the previous section, client-side security is gaining more and more importance. Therefore, new web security mechanisms like the Content Security Policy are developed to further strengthen the security level on the client-side. However, the issue of client-side security is far away from being solved.

To better understand the need for an advanced web security concept, in this section the main problem from the perspective of a trustworthy web service is described. From the viewpoint of the web service, the client-browser can be seen as a **blackbox** (see figure 6). The **blackbox** assumption is made due to the following three reasons:

Figure 6: The current problem from the perspective of the web service

- **Missing identification:** Most web security related web services are using HTTPS (with SSL certificates) to secure their network connection with the requesting user. Based on the received information within the certificate, the user is able to clearly identify the web service (see figure 7 for an advanced EV-SSL certificate). From the perspective of the web service, it is unable to check the identity of the user. Therefore, the web service cannot determine if the web request is sent by the intended user or by an adversary impersonating the user on the client-

side. Without an explicit and secure authentication/ identification mechanism, the web service is unable to check the identity of the requesting user.

- **Manipulated information:** All information sent by the client-browser, can easily be manipulated on the client-side (for example by a Man-In-The-Browser) or during the transmission of the web packets (for example by a Network-Attacker). To increase the security on the client-side, the web service uses client-side information to adapt the security rules individually and can determine possible security vulnerabilities on the client-side. As the transmitted information can be manipulated by an adversary, the web service cannot trust the information sent by the client-browser. This further decreases the client-side security.

- **Enforcing security rules:** As the web service cannot determine if the sent web packets are unchanged (see manipulated information), it is possible that security related information (for example security rules) are not evaluated on the client-side. Even though a the web service has specified client-side security rules, which are increasing the security level and therefore mitigating several client-side attacks, the web service is unable to check if these rules are truly enforced. Therefore, an adversary, who manipulates the client-side environment or the web packets during the transmission over the network, is able to block the enforcement of security rules. Again, the web service has to rely on the client-browser that all rules are implemented.

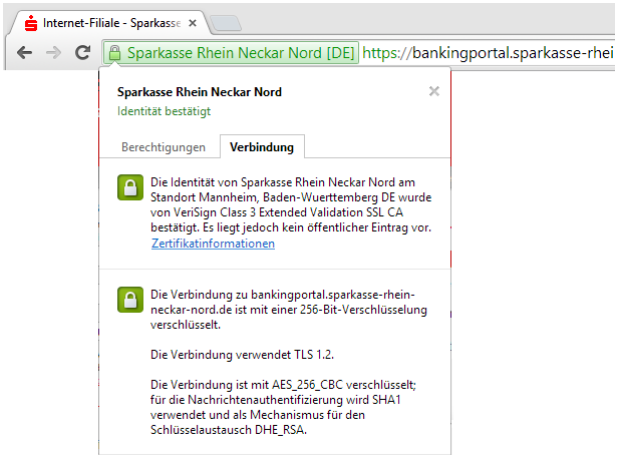

Figure 7: EV-SSL certificate information on the client-side

Seen in the previous listing, the current client-side security concept from the perspective of the web service is based on trust. The web service has no possibility to get reliable information about the execution of security related tasks at the client-browser. Therefore, we make the assumption, that without solving the issues of **explicit client-side identification**, **providing a mechanism to collect correct client-side information** and **the ability to enforce client-side web security rules**, maximum client-side security is not possible.

Therefore, the following work introduces a new advanced web security concept called **Secure Session Protocol**, which enables the web service to create individual client-browser security rules based on correct information about the client-browser. Furthermore, the web service is able to enforce them with the help of a trustworthy partner on the client-side. The property of client-side identification is achieved by design.

Before continuing with the detailed explanation of the Secure Session Protocol, we take a closer look at an example, which is able to enforce security policies on the client-side.

The **Browser-Enforced Embedded Policy (BEEP)** [31] is another client-side web security policy, which tries to mitigate Cross-Site-Scripting (XSS) attacks by controlling *inline JavaScript execution*. Through input fields at the web service, an adversary might be able to inject arbitrary code into the web service. To avoid the malicious behavior, the web service needs to filter untrusted HTML input. By using the BEEP, the web service developer is able to specify a whitelist, which contains all inline scripts of the web service. If the user executes script code on the web service, the script is compared with the list before its execution. If the script is not listed in the whitelist, the execution of the script code is denied by the BEEP.

The **Content Security Policy**, developed by the Mozilla Foundation, introduces a client-side web security policy [40], which is delivered by the web service through an HTTP header field. The web service specifies location URL's from which additional resources (scripts, images or media files) are allowed to be loaded by the client-browser. Further, by providing several key words, the web service is able to regulate the execution of inline JavaScript and the usage of the JavaScript function *eval*. By definition, the Content Security Policy denies all resource requests by the client-browser. Inline JavaScript and the execution of the function *eval* is not allowed by default. Therefore, the web service has to explicitly specify allowed operations by the client-browser.

The **Security Style Sheet (SSS)** [60] is another client-side web security concept. It is based on Security Style Sheets (SSS), which are defining rules for single web page elements. Similar to the Cascading Style Sheets (CSS) [67], the web service developer is able to define a security rule for a single web page element (id) or for a class of web page elements (class). Within the SSS it is possible to define a whitelist of allowed locations for loading external resources. Furthermore, the web service developer can specify the execution of script code within the web page element. In Contrast to the other two mentioned web security policies, the SSS further regulates the communication between the single web page elements. Within the SSS the web service developer is able to define a list of allowed web page elements, the current web page element is allowed to communicate with. As mentioned in the beginning of this paragraph, the SSS is currently only a theoretically web security concept.

The three mentioned client-side web security policies are all delivered by the web service. The web service relies on the client-browser, that the web security policy is enforced on the client-side. Today, the web service has no possibility to verify that the client-browser has enforced the web security policy. The last example for an advanced client-side security mechanism introduces a concept, which in contrast to the Secure Session Protocol and the three mentioned client-security rules, secures the complete client-computer instead of only the client-browser.

The software product **Endpoint Security VPN** by Check Point [7] increases the client-side security level by enforcing security policies at the client-computer. The software is used to connect the client-computer through a VPN tunnel to a company network. In general, client-computers connecting to secure company networks have to be compliant to the company policies. The software first starts with downloading the company security policy from the policy server through a pre-established VPN connection. Then, the Endpoint Security VPN software enforces the security policies of the company on the remote client-computer. Thus, the client-computer is then allowed to connect to the secure company network.

Endpoint Security VPN in contrast to the Secure Session Policy, regulates and secures the complete client-computer as in contrast the Secure Session Protocol only operates at the client-browser. Further, Endpoint Security VPN enforces the policy permanently. Thus, the functionality of the client-computer might be limited. The security policy is enforced on the client-computer as soon as the Endpoint Security VPN client has downloaded the security policy from the policy server. According to the permissions of the user within the company network, the permissions of the user on the remote client-computer might be degraded. For example, the client-user might not be able to install additional software or modify system properties. This heavily decreases the usability of the client-computer for the private usage of the computer.

Due to the previously described problem, Endpoint Security VPN is currently only used for business applications as it additionally requires infrastructure to be previously set up. In the following chapter we describe the Secure Session Protocol, which combines the security features of the presented solutions and eliminates the mentioned drawbacks.

3 Concept

The main chapter of this work introduces the concept of the **Secure Session Protocol**. The Secure Session Protocol is an advanced web security protocol, which tries to maximize the security level on the client-side. Further, the overall security of the web service is increased. The Secure Session Protocol enables the web service to execute the following listed mechanism:

- **Collecting correct information:** The Secure Session Protocol enables the web service to receive correct information about the client-browser. Current web services cannot verify, if the received data has been manipulated on the way to the web service.
- **Explicit identification:** If the Secure Session Protocol is executed properly, the client and the web service can exactly identify the opposite communication partner.
- **Enforcing security rules:** The web service is able to enforce web security rules at the client-browser. Web security rules are enabling different web security mechanism at the client-browser, e.g., regulating the execution of script code. Therefore, it is important for the web service to be sure about the implementation of sent web security rules.
- **Detecting security issues:** While the web session (throughout the work we call an established web session through the Secure Session Protocol a Secure Session) is active, the Secure Session Protocol is able to detect client-side attacks.
- **Detection of outdated software on the client-side:** During the Establishing state (see subsection 3.3.3), the web service is able to check if the software requirements on the client-side are met. The security checks are based on correct information collected on the client-side. For example, an outdated browser version can lead to an increased security risk.

The rest of this chapter describes the Secure Session Protocol execution and explains, how the above mentioned mechanism are achieved. First, an overall protocol description of the main concept is given. Then, the individual protocol participants are presented and their main tasks are described. The next section gives an overview of the complete protocol and explains all states of the Secure Session Protocol in detail. Newly introduced terms and concepts of the protocol are defined when first used within the section. An example of the Secure Session Protocol execution is given in chapter 5. The example includes a detailed description of all relevant Secure Session Protocol steps. This work further includes a concept implementation of the Secure Session Protocol. The programming part of this work is explained in chapter 6.

3.1 Protocol Participants

In total, the Secure Session Protocol has three protocol participants: the **browser of the user**, the **Secure Session Protocol Extension** and the **web service**. In the following listing these participants are described and the dependencies are visualized in figure 8:

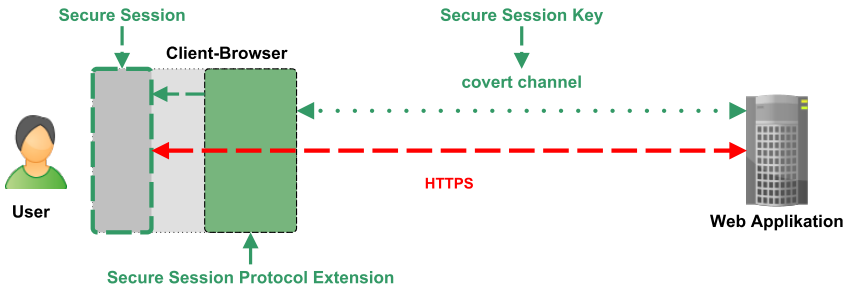

Figure 8: The Secure Session Protocol

- **Client-Browser**: The **Browser (client-browser)** describes the user-agent installed on the client-computer. With the client-browser, the user is able to request content from a web service. During the Secure Session Protocol, it has no additional responsibilities. The implementation described in chapter 6 is based on Google Chrome [22]. Nevertheless, the Secure Session Protocol can possibly be implemented in any current browser. As described in the previous section 1.2, we consider that the base installation of the browser is correct and has not been modified by an attacker previously.

- **SSP-Extension**: The **Secure Session Protocol Extension (SSP-Extension)** is responsible for the correct execution of the Secure Session Protocol on the client-side. The SSP-Extension is a **trustworthy partner** of the web service on the client-side. A correct installation is an essential condition to execute the protocol in a secure manner. This problem is further discussed in chapter 4. An example of an implementation of the SSP-Extension is given in chapter 6. The different responsibilities and functions of the SSP-Extension are described throughout the following chapter.

- **Web service**: The requested content by the client-browser is provided by a **web service**. A web service is hosted on a web server, which is reachable through the intra- / internet. In order to execute the Secure Session Protocol, the web service communicates with the SSP-Extension on the client-side directly.

3.2 General

The chapter continues with a short overview of the complete Secure Session Protocol. Then, a detailed explanation of the single states are given. Figure 9 visualizes the connections of the single states of the Secure Session Protocol. From section 3.3.1 to 3.3.6 the single states are described and the requirements for a transmission are discussed.

The Secure Session Protocol starts after the SSP-Extension has been successfully installed at the client-browser. The SSP-Extension starts the execution by entering the **Ready** state. The SSP-Extension signalises any requesting web services that the client-browser is able to handle the Secure Session Protocol. If a web service is able to communicate through the Secure Session Protocol, it adds additional information to the web response to inform the SSP-Extension on the client-side that the web service supports the Secure Session Protocol. Further, if the web service and the SSP-Extension have executed the Secure Session Protocol in the past, the SSP-Extension has locally stored pairing information. Depending on the identification result, the state of the protocol switches into the state **Establishing** or **Pairing**. Before a Secure Session for a specific web service can be established for the first time, a Secure Session Key, unique for the pair of SSP-Extension and web service, needs to be created and exchanged. The creation of the Secure Session Key is based on the Diffie-Hellman Key Agreement [42]. To mitigate Man-In-The-Middle attacks, an Out-of-Band secret is used, to verify the identity of the web service. The Secure Session Key is then stored on both sides. If a valid Secure Session Key is available in the local storage of the SSP-Extension, the protocol skips the **Pairing** state and continues with the **Establishing** state of the protocol. The **Establishing** state is further divided into three sub-states **Collecting**, **Creating** and **Building**. The main purpose of the first part of the **Establishing** state, is to collect correct information about the client-browser (Collecting). Based on the collected information by the SSP-Extension, the web service creates an individual Secure Set for the client-browser (Creating). Appropriate to the Secure Set specified by the web service, the SSP-Extension builds the Secure Tab (Building). Within the Secure Tab, all security parameters and rules are enforced. The protocol enters the **Running** state by starting the Secure Session in the Secure Tab. The SSP-Extension takes care of the enforced Secure Set during the **Running** state. After the Secure Session is terminated, the protocol terminates by entering the **Terminating** state. In this state of the Secure Session Protocol, the SSP-Extension resets all protocol parameters. The **Terminating** state of the protocol is also reached, if during all states of the protocol (except **Ready**) one of the communication partners encounters a protocol or security issue. To be able to execute a new Secure Session, the protocol switches into the start state **Ready**.

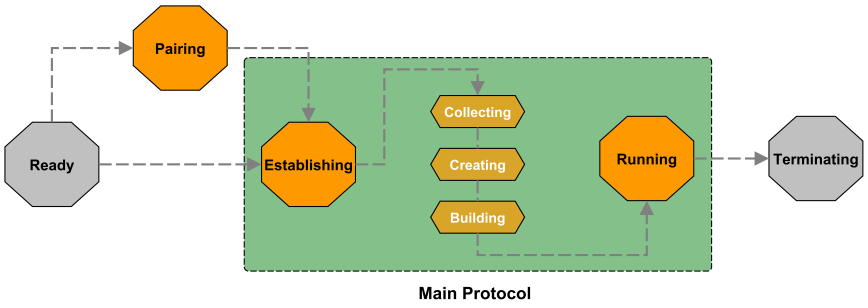

Figure 9: The simplified Secure Session Protocol

Shown in figure 10, the protocol participants can signalize a transition from one state into another by sending predefined values to the opposite communication partner. The values are sent through an HTTP header field. The field has the name *Secure-Session-Protocol* (usually unstandardized HTTP header fields are using an X-prefixing, RFC 6648 [28]

describes that this technique is deprecated and therefore the HTTP header for the Secure Session Protocol is simply the name of the protocol). Specified in Figure 9, only valid transitions can be made by either communication partner. To simplify this technique for the reader, in the remainder of this chapter, setting the HTTP header field is simply expressed through the *value of the current Secure Session Protocol state*.

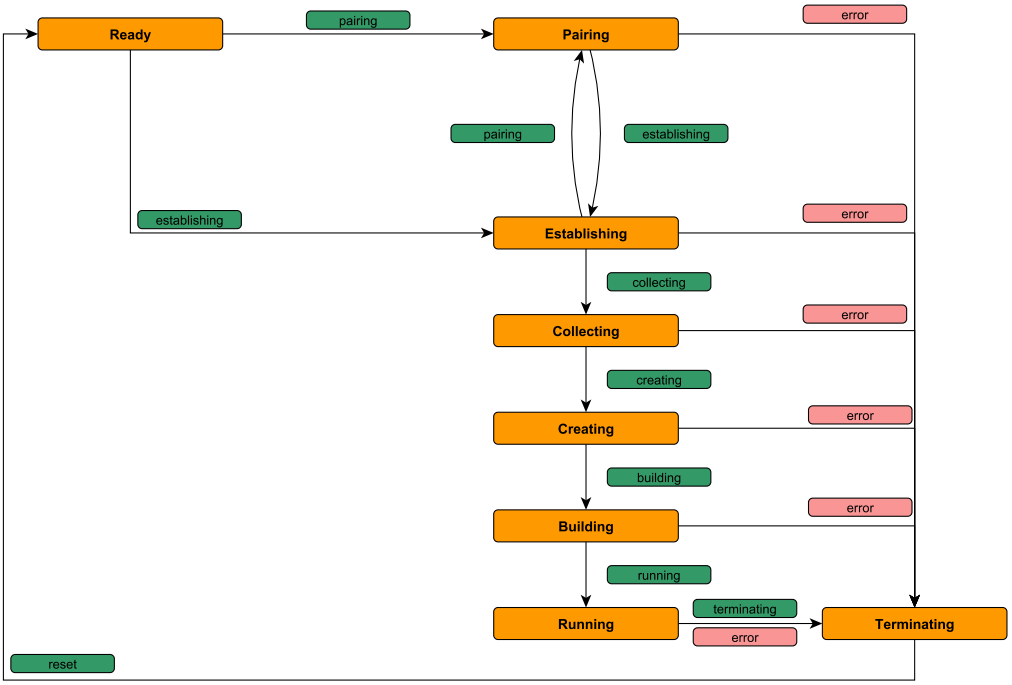

Figure 10: The complete Secure Session Protocol, including errors and transition values

3.3 Protocol Overview

The following section is divided into five subsections. Each of them describes one single state of the Secure Session Protocol. To get a quick overview of the currently explained state, each subsection starts with an short overview including possible predecessor and successor states and a short description of the main task of the current state:

Example	
Predecessor states	Includes all possible predecessor states, from which the Secure Session Protocol can change into this state with **one** transition.
Transition values	Covers all values, which the Secure Session Protocol HTTP header can carry to change the current state of the Secure Session Protocol.
Successor states	Contains all states that can be reached through **one** possible transition, starting in the current state of the Secure Session Protocol.
Description	Gives a short description of the current state, including the intended purpose of the state and the meaning for the whole Secure Session Protocol.

3.3.1 Ready

Ready	
Predecessor states	$P = \{\emptyset, Ready, Terminating\}$ (1)
Transition values	$\delta : \{initialization, pairing, establishing\}$ (2)
Successor states	$S = \{Ready, Pairing, Establishing\}$ (3)
Description	The **Ready** state is the initial state of the Secure Session Protocol. The SSP-Extension has to determine if a Secure Session is possible to establish with the requested web service.

The main task of the first state of the protocol is to discover, if the requested web service supports the Secure Session Protocol and thus is able to communicate with the SSP-Extension. The described behavior of the SSP-Extension during the **Ready** state is shown in figure 11.

The SSP-Extension has locally stored information about *already paired web services*. If the web service and the SSP-Extension have previously executed a Secure Session, they share the same pairing information. Hence, the SSP-Extension knows that the web service will try to establish a Secure Session on the client-side. The protocol continues with the **Establishing** state. If not, the web service is unknown to the SSP-Extension (no pairing information are shared). Therefore, the SSP-Extension signalizes the requested web service that the client-browser is able to execute the Secure Session Protocol. If the receiving web service is able to communicate through the Secure Session Protocol, the protocol will switch to the **Pairing** state. As long as the extension does not detect a suitable web service, the Secure Session Protocol remains in the **Ready** state.

In total, three possible successor states, namely **Ready, Pairing** and **Establishing**, as shown in Figure 11, are possible. To evaluate which of those states is executed next, at most one request from the client and one response from the server are required. The next three paragraphs explain in detail, which prerequisites have to be fulfilled to make a transition into one of the three successor states.

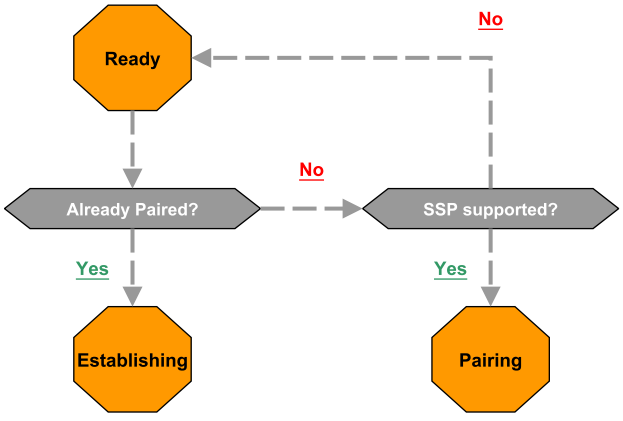

Figure 11: Possibly successor states of the Ready state

Ready to Establishing

The SSP-Extension stores information about already paired web services (further information see 3.3.2). *Known* web services have exchanged all needed information with the SSP-Extension to create a Secure Session on the client-side. This information is called *pairing information* and is exchanged during the **Pairing** state. The SSP-Extension starts to establish the Secure Session with the already paired web service.

As seen in section 3.3.3, the SSP-Extension alone cannot determine if the protocol changes to the **Establishing** state as it might be possible that the web service wants to renew the pairing information or that the web service does not longer support the Secure Session Protocol. Therefore, the SSP-Extension cannot change the protocol state alone.

The following transitions of the protocol occur, if the web service is requested for the first time (the web service is *unknown*). Hence, the SSP-Extension does not know, if the web service is able to execute the Secure Session Protocol. In this phase of the **Ready** state, the SSP-Extensions needs to decide whether the requested web service supports the Secure Session Protocol or not. The SSP-Extension adds an HTTP header to the original web request, indicating that the client-browser is able to execute the Secure Session Protocol. As mentioned in the previous section, the HTTP header field *Secure-Session-Protocol* is used. The value is set to *initialization*.

Ready to Pairing

If the web service has implemented the Secure Session Protocol, the protocol switches to the **Pairing** state after receiving the value *initialization*. The web service sets the Secure Session Protocol value to *pairing*, indicating the SSP-Extension that the protocol switches to the state **Pairing**.

At this point of the protocol execution, the web service and the client-browser both supporting the Secure Session Protocol. To establish a Secure Session on the client-side, they need to the share the same Secure Session Key (part of the pairing information). This key is created and exchanged during the **Pairing** state. Further information on the **Pairing** state and the purpose of the **Secure Session Key** is given in section 3.3.2.

Ready to Ready

If the web service does not support the Secure Session Protocol, the web service simply ignores the HTTP header value *initialization*. The original request of the client-browser is processed and the requested content is delivered as usual. In this case, no transition is done. The Secure Session Protocol remains in the **Ready** state. If the web service is already paired with the SSP-Extension, but does not respond correct, accordingly to the Secure Session Protocol specification, it might be possible that an adversary modifies the network connection. In this case, the SSP-Extension displays a warning on the client-side, because a potential security issue is found.

As an important side note, if a web service does not support the Secure Session Protocol the web service is delivered as usual. Besides the added HTTP header field no additional overhead is created.

The Concept chapter now continues with the description of the required **Pairing** state of the Secure Session Protocol. Throughout the rest of this chapter, we assume that the web service supports the Secure Session Protocol.

3.3.2 Pairing

Pairing	
Predecessor states	$P = \{Ready, Establishing\}$ (4)
Transition values	$\delta : \{establishing, error\}$ (5)
Successor states	$S = \{Establishing, Terminating\}$ (6)
Description	To create a Secure Session on the client-side, a unique Secure Session Key needs to be shared by both communication partners. The key is unique for the web service and the SSP-Extension. A Secure Session on the client-side is only possible, if the web service and the SSP-Extension have previously exchanged a Secure Session Key.

After the Secure Session Protocol switches from the state **Ready** to the state **Pairing**, both are sure that a Secure Session can be established. For establishing a Secure Session each pair of communication partners, composed of a client-browser with a SSP-Extension and a web service, need to do the Pairing phase in advance. The main purpose of the **Pairing** state is to create and exchange the Secure Session Key.

At this stage of the Secure Session Protocol, the client-browser cannot confirm the identity of the web service. If the web service is using HTTPS, the user can verify the identity of the web service by checking the delivered HTTPS certificate. Nevertheless, as mentioned in the introductory chapter of this work, it might be possible for an adversary to spoof the HTTPS connection (SSL-Stripping attack). The Secure Session Protocol introduces the concept of the Secure Session Key. During the creation of the Secure Session Key, the user explicitly checks the identity of the web service.

Definition. *The **Secure Session Key** is a symmetric key used for encrypting all messages sent between the web service and the SSP-Extension. The Secure Session Key is unique for the combination of web service, user-ID and SSP-Extension. The created key is stored together with the user-ID (unique identifier of the user for the web service) on both sides in a secure environment. Therefore, it is possible for different users to establish a Secure Session with the same client-browser. The exchanged Secure Session Key is valid until one communication partner initiates a new pairing and therefore revokes the shared key.*

*The **Secure Session Key** is used for establishing a covert channel between the web service and the SSP-Extension. Further, the web service uses the Secure Session Key to identify the user at the client-side. Both functions of the Secure Session Key are further explained in chapter 4.*

The **Pairing** state can be reached by two states of the protocol. As described in the previous section the purposed predecessor state is **Ready**. If the SSP-Extension sends the *initialization* value to the web service, it initiates a new pairing. The second possibility is that the web service wants to renew the pairing information. Stated in the description of the Secure Session Protocol, the pairing state is executed only once. Nevertheless, it is possible for both communication partner to initiate a new pairing phase. In this phase of the protocol, the web service can set the Secure Session Protocol value to *pairing*. The state of the protocol switches back from **Establishing** to **Pairing**.

The renewal of pairing information can be due to security issues on either sides or because of policy requirements of the web service. Further, some web services might want to execute the **Pairing** state for each Secure Session. Then, each time a Secure Session is established, a new Secure Session Key is used. Therefore, the usage of the Secure Session Key is limited. This improves the security level of the Secure Session further. On the other hand, the user experience might be decreased, as the **Pairing** state creates additional overhead.

If the SSP-Extensions should renew the pairing information, the user needs to delete the locally stored pairing information. Thus, the requested web service is unknown to the SSP-Extension and the **Pairing** state is executed again. Therefore, only in the case of a renewal initiate by the web service, a transition from **Establishing** to **Pairing** is possible (If the user deletes the pairing information, the protocol switches from Ready to Pairing).

The section continues with a detailed description of the pairing protocol. The single steps, including a list of the sent parameters, are explained.

Pairing Protocol

To create the Secure Session Key a Diffie-Hellman Key Agreement (DH protocol) [42], with an Out-of-Band secret to verify the identity of the web service, is used. In both cases, which are leading to a pairing (Ready to Pairing / Establishing to Pairing), the first step of the DH protocol is executed by the web service. The protocol execution is visualized in 12. The parameters in grey are public, the red ones are private. The pairing protocol has five steps, which are explained in detail in the following paragraph:

***Note:** The choice of the security parameters of the Diffie-Hellman Key Agreement are discussed in chapter 4. Both communication partners in advanced have exchanged the necessary parameters p (public prime number used as the size of the cyclic group) and g (public prime base, generator of the group p). By definition described in [42], both parameters can be public.*

- **Step 1:** The web service starts the pairing protocol by calculating the server side secret, which is the first part of the Secure Session Key. The web service creates a large random prime number a and calculates:

$$A = g^a \bmod p \tag{7}$$

$$V = E(s, pk) \tag{8}$$

The result A (first part of the Secure Session Key) is transmitted to the SSP-Extension on the client-side. To avoid Man-In-The-Middle attacks, the SSP-Extension needs to verify that the value A is created by the web service. Therefore, the web service encrypts the public key hash pk of his own HTTPS certificate with a symmetric encryption method $E(key, plaintext)$ (e.g. AES-256) by using an additional secret s as the key. The encrypted public key

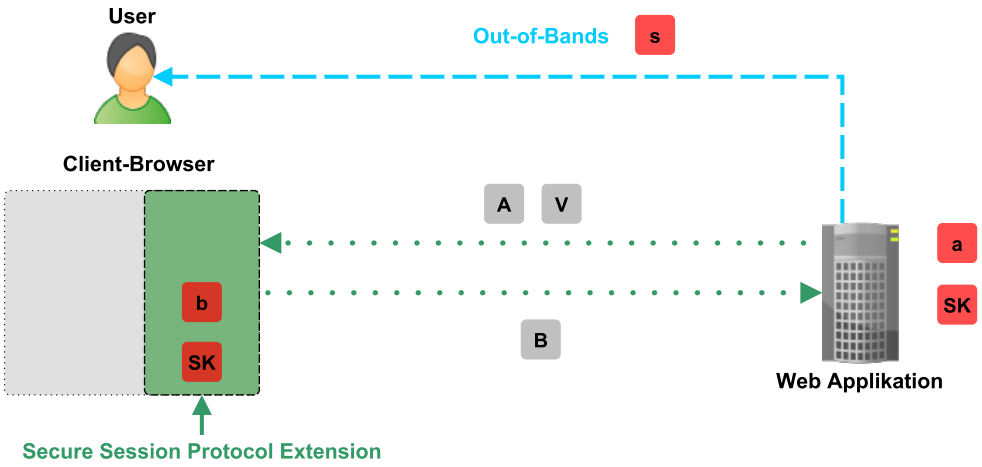

Figure 12: The setting of the pairing protocol

hash V, is sent together with A back to the SSP-Extension. The secret s is delivered to the user over an Out-of-Band channel.

Note: *We assume that the Out-of-Band channel is secure and not compromised by an adversary, who cooperates with the other adversary, who currently compromises the HTTPS connection between the web service and the SSP-Extension.*

- **Step 2:** The SSP-Extension receives the values A and V from the web service. First, the SSP-Extension verifies that the received values are sent by the web service. The public key hash pk' is extracted from the HTTPS certificate of the web service. Then, the value V is decrypted with the symmetric decryption function $D(key, cipher)$ by using the Out-of-Band secret s as the key. The output of the function D and the extracted public key hash of the HTTPS certificate are compared. If the values are identical, the SSP-Extension can be sure that the transmitted parameters are from the intended web service. The SSP-Extension continues with creating the Secure Session Key SK and finally calculates the DH value B (second part of the Secure Session Key) for the web service. The SSP-Extension starts creating a large random prime number b and calculates:

$$pk = D(s, V) \tag{9}$$

$$pk == pk' \tag{10}$$

$$B = g^b \bmod p \tag{11}$$

$$SK = A^b \bmod p \tag{12}$$

B is sent back to the web service. The Secure Session Key SK is stored together with the user-ID and the web service URL within a secure storage environment of the client-browser.

- **Step 3:** The web service receives the value B from the SSP-Extension. Thus, the web service is able to calculate the Secure Session Key SK on its own:

$$SK = B^a \bmod p \tag{13}$$

After calculating the Secure Session Key, the web service saves the symmetric Secure Session Key together with the user-ID on the web server.

All intermediate used secrets, a and b, are deleted after the Secure Session Key is calculated. This is necessary to achieve Perfect Forward Secrecy, as described in [37].

The attentive reader might wonder, why the web services does not check for any Man-In-The-Middle attacks. If step three of the pairing protocol is compromised by an adversary and the sent value B is exchanged, the Secure Session Key on both sides is not equal. As the following communication between the web service and the SSP-Extension is encrypted with the Secure Session Key, the Secure Session Protocol would terminate in the next state immediately, because the communication partners are not able to decrypt necessary information. This failure would lead to a termination of the protocol, followed by a new pairing protocol execution.

If the pairing has been successfully finished, both communication partners share the same symmetric Secure Session Key and the **Pairing** state is done. The Secure Session Protocol switches into the **Establishing** state.

3.3.3 Establishing

Establishing	
Predecessor states	$P = \{Ready, Pairing\}$ (14)
Transition values	$\delta : \{collecting, pairing, error\}$ (15)
Successor states	$S = \{Pairing, Collecting, Terminating\}$ (16)
Description	The SSP-Extension builds the Secure Tab on the client-side accordingly to the Secure Set, which is specified by the web service based on correct information collected by the SSP-Extension.

The main task of the **Establishing** state is to establish the Secure Session on the client-side. The **Establishing** state is further divided into three phases, **Collecting, Creating** and **Builidng**. The **Collecting** phase, executed by the SSP-Extension, collects correct information about the client-browser and the client-side environment. The web service continues with the **Creating** phase, by creating the Secure Set. The Secure Set specifies security rules and parameters, which are determined by the correct information collected during the **Collecting** phase. The **Establishing** state concludes with the **Building** phase. The Secure Tab is built on the client-side by the SSP-Extension. All specified security rules (Secure Set) are enforced within the Secure Tab.

In the first part of this subsection, definitions of the required concepts for establishing a Secure Session are given. Followed by general conditions for entering the **Establishing** state, the subsection continues with a detailed description of the three sub-phases.

Definition. *The **Secure Set** of the Secure Session Protocol is a data structure consisting out of security parameters and security rules. The Secure Set is created by the web service. The security parameters and security rules are enforced on the client-side by the SSP-Extension within the Secure Tab. The values are individually specified by web service for each Secure Session. According to the current specification of the Secure Session Protocol, the web service has to define six different parameters and rules (further described in section 3.3.3).*

Definition. *The **Secure Tab** is the graphical element at the client-browser for the Secure Session. The Secure Tab is a separate browser window, created by the SSP-Extension. Appropriate to the Secure Set, the Secure Tab enforces all security parameters and security rules received from the web service. The Secure Tab indicates to the user the lifetime of the Secure Session. A Secure Session starts with the opening of the Secure Tab and is terminated if the Secure Tab is closed.*

The **Establishing** state of the protocol is reached if both communication partners share the same Secure Session Key. This is only the case, if both have executed a successful pairing in advanced. Hence, this state is reached after the **Pairing** state or if the web service was paired before (transition from **Ready** to **Establishing**).

In the following subsections of this chapter, the three phases of the **Establishing** state are explained.

Collecting	
Predecessor states	$P = \{Establishing\}$ (17)
Transition values	$\delta : \{creating, error\}$ (18)
Successor states	$S = \{Creating, Terminating\}$ (19)
Description	The SSP-Extension collects correct information about the client-browser and the client-side environment, which are then encrypted and sent to the web service.

Defining the Secure Set is essential for the later achieved security at the Secure Tab, as described in section 4. For building a proper Secure Set, the web service needs correct and reliable information about the client-browser. To ensure the correctness of the collected information, the SSP-Extension, as a trustworthy partner of the web service, extracts required information from the client-browser and encrypts the data with the shared Secure Session Key, before sending the web packet back to the web service.

In the current draft of the Secure Session Protocol, five properties are collected by the SSP-Extension. This information is used to create the Secure Set, but is further used for evaluating if a Secure Session is possible to be established. Therefore, initial security checks are executed by the web service during the **Creating** phase. The following listing describes all properties collected by the SSP-Extension and its intended purpose:

- **HTTP Header Field Host:** The mandatory HTTP [43] header field *Host* is extracted from the original web request by the client-browser. It describes the destination URL of the requested web resource. The *Host* field is used for the Secure Set only.
- **User Agent:** The User Agent is an HTTP header field, which contains information about the used client-browser. This includes browser vendor, browser version and operating system. The SSP-Extension extract this information directly from the browser, building the *User Agent* string on its own. The *User Agent* is used for the security checks and the Secure Set.

  Note: Difference between Extension and Plug-In: *To fully understand the security model, it is highly relevant to mention the difference between extensions and plug-ins. In the following sections a custom browser extension describes a software solution, which can be manually installed into the client-browser. The origin of the custom browser extension is not defined and not need to be trustworthy. By definition a custom browser extension is not mandatory for executing a web service correctly. On the other hand, a browser plug-in is a software module distributed by a trusted third-party, which can be necessary for finite web services. Custom browser extensions can only access client-browser related data. In contrast, plug-ins can (depending on their permissions) access the underlying operating system.*
- **Active Browser Extensions:** Current browser vendors (e.g. Mozilla and Google) are supporting the possibility to extend the functionality of the client-browser with *custom browser extensions*. The SSP-Extension checks, if custom browser extensions are installed and running. This information is used for the Secure Set.
- **Installed Browser Plug-in Versions:** The SSP-Extension creates a list of all installed browser plug-ins together with the current software version. This information is used for the security checks and the Secure Set.
- **Network Status:** Additional to the browser specific information, the SSP-Extension extracts information about the network status of the browser. If the current connection is established through a proxy server [2], this information is sent to the web service. This property is used for the previously executed security checks.

Any collected information is packed together with the Secure Session Protocol value *collecting* and is encrypted with the Secure Session Key.

[2] A transparent proxy cannot be detected by the SSP-Extension

Properties of the original web request are duplicated in the encrypted packet as they are anyway present in the HTTP request. The HTTP header fields *Host and User-Agent* can be found twice in the web request. The collected properties are relevant for the web service. They are used for the security checks and the Secure Set. Thus, the information need to be correct and are therefore collected by the SSP-Extension (trustworthy partner) and encrypted with Secure Session Key. The security mechanism of sending data redundant, is further explained in chapter 4.

Section 6.4 describes an approach, which enables the web service to specify the information collected by the SSP-Extension. The list of collected information is then stored within the local storage of the SSP-Extension on the client-side. How this technique on the one hand can reduce the overhead and runtime of the Secure Session Protocol and on the other hand might lead to a privacy problem, is described in the chapter 7.

Creating

Collecting	
Predecessor states	$P = \{Collecting\}$ (20)
Transition values	$\delta : \{building, error\}$ (21)
Successor states	$S = \{Building, Terminating\}$ (22)
Description	The web service executes security checks based on the correct information received by the SSP-Extension to evaluate if a Secure Session is possible. The web service then continues the Establishing state by creating the Secure Set.

After receiving and decrypting the web packet from the SSP-Extension, the web service continues with the execution of the *Creating* phase. At the beginning of the **Creating** phase the web service performs initial security checks for evaluating, if a Secure Session is possible. The evaluation is based on the information sent by the SSP-Extension in the *Collecting* phase beforehand. If the web service concludes, that a Secure Session is possible, the next task of the web service is to create the Secure Set which is used for building the Secure Session in the *Building* phase.

Performing initial security checks

Described in chapter 4, it is crucial for the security of the established Secure Session to create an appropriate Secure Set. Nevertheless, certain prerequisites have to be met to be able to establish a Secure Session. The security checks are required to determine, if the current security level of the client-side environment is suitable for establishing a Secure Session. If the prerequisites (individually defined by the web service) cannot be fulfilled by the client-browser, the web service terminates the Secure Session Protocol in the **Creating** state of the protocol.

This step is optional for the web service but recommend, because it increases the security level of the Secure Session enormously. As it is a security concept for the web service provided by Secure Session Protocol, this technique is not further discussed and specified as the main focus of this work is on client-side security. Nevertheless, a quick overview of possible checks are shown:

- **Examine the User Agent:** The web service can decide to exclude users with outdated browser versions by checking the user agent. If the web service retrieves a client-browser with an outdated browser version, the Secure Session is terminated and the user is informed.
- **Checking Plug-In Versions:** Outdated software components can be exploited by an adversary. Thus, checking the version of installed plug-ins enables the web service to detect possible vulnerabilities on the client-side.
- **Comparing redundant information:** The web service can compare the correct information by the SSP-Extension (collected during the **Collecting** phase) with the original information from the web request by the user. This enables the web service to detect modifications of the web packets.

In section 6.4, an advanced mechanism for collecting client-side information is introduced as mentioned in the previous section. Further, depending on the implementation of the SSP-Extension, the web service is able to check client-side environment properties, e.g., from the underlying operating system.

Definition. *During the execution of the Secure Session Protocol* ***warnings and errors*** *can occur, which can lead to a termination of the protocol. This is the case, if the web service or the SSP-Extension discovers abnormal behavior on either side. In the case of an error, the Secure Session Protocol terminates immediately. By encountering a warning the Secure Session Protocol continues, by reporting the warning to the user and the web service. If a warning or an error occurs, the user at the client-browser is informed with a message. In the case of an error, which leads to a termination, the value of the Secure Session Protocol is set to termination. To be safe against any kind of manipulation warning and error messages are always sent encrypted between the two communication partners. One goal of the Secure Session Protocol is to retrieve possible security vulnerabilities on the client-side, which are then reported to the user (an example is shown in the warning example).*

- **Example Warning:** The requested web service requires a third-party browser plug-in for a correct execution. Based on the collected information by the SSP-Extension, the web service discovers that the installed plug-in version is outdated, but is however runnable. Therefore, the web service adds a warning informing the user that the plug-in version should be updated, because there is a potential security risk, by using outdated browser plug-ins [53].
- **Example Error:** The HTTP header field *User-Agent*, collected during the *Collecting* phase of the **Establishing** state, includes information about the used browser version on the client-side. Depending on the security requirements of the web service, it is possible that the Secure Session Protocol terminates due to an outdated (therefore insecure or not compatible) browser version. The SSP-Extension is informed about the potential security vulnerability, displaying the user the error message with the hint to update the browser version, to be able to execute the web service.

As the security checks are web service specific, a description of a possible evaluation of the sent information is given in chapter 5 as an example. If the web service determines an error during the checks, the Secure Session Protocol terminates in this phase of the **Establishing** state. Otherwise, if no errors have occurred and all security checks are successfully executed, the protocol continues with the creation of the Secure Set. Raised warnings during the initial security checks are added to the response for the SSP-Extension.

The current Secure Set consists out of five security parameters and one security rule. All parameters and rules of the Secure Set are mandatory. The possible values for the security parameters and rules are given in the brackets, following the value name:

Parameter

- **Private Browsing (true/false):** The parameter *Private Browsing* specifies if data created during the Secure Session, is stored or deleted after the Secure Session is terminated. The data generated by the web service, includes visited web pages, form and search bar entries, passwords, downloads, download list entries, cookies and cached data (web and off-line content). The private browsing mechanism is further described in [41]. To enable private browsing the parameter is set to *true*, else to *false*. See chapter 6 how this security feature is currently executed.
- **Extensions (true/false):** By setting the *Extensions* parameter, the web service is able to specify if custom browser extensions should be disabled throughout the established Secure Session. If the parameter is set to *true*, all custom browser extensions are disabled. For no changes, the parameter is set to *false*. In [3] the possible security impact of custom browser extensions is described.
- **Plug-Ins (true/false):** Similar to the *Extensions* parameter, the *Plug-Ins* parameter manages installed third-party plug-ins. By setting the parameter to the value *true*, all third-party plug-ins are disabled throughout the Secure Session. If the value is set to false no changes to installed plug-ins are done.
- **Entry Point (URL scheme):** The *Entry Point* parameter specifies the URL for the first web request of the client-browser to the web service after the Secure Session is established. An URL String specifies the value of the parameter. The HTTP header field host is overwritten by the *Entry Point* value later in the protocol (see section 3.3.5).
- **HTTPS Encryption (public key/false):** If the web service uses HTTPS, the *HTTPS Encryption* parameter can be used to ensure that a secure HTTPS connection is established on top of the Secure Session Protocol. If the parameter is set to false, no additional methods for securing the HTTPS connection are done. If set to true, the parameter contains the public key hash of the HTTPS server's certificate as its value. The SSP-Extension further controls, if the used protocol is always HTTPS and the checks the public key hash of any established connection.

Rules

- **Content Security Policy (CSP syntax / {}):** The *Content Security Policy* rule specifies a set of key-pair values, which enables the client-browser to control the locations from where different types of content are allowed to be loaded [40]. A complete description of all functionalities of the Content Security Policy is given in chapter 4. By definition, the Secure Session Protocol uses the current draft of the Content Security Policy version 1.1. from the 11th February 2014 [1]. The Syntax of the Content Security Policy is defined by the W3C and is not part of this work. If the policy is not defined, the value of this policy is set to {}.

The finally created Secure Set is encrypted with the Secure Session Key and send back to the SSP-Extension. Warnings are added to the response as well as the Secure Session Protocol value *building*. If the Secure Session Protocol terminates with an error, due to failures during the security checks, the web service responses with the Secure Session Protocol value *terminating* and an encrypted error message, which carries the error raised during the **Creating** phase.

3.3.4 Building

Building	
Predecessor states	$P = \{Creating\}$ (23)
Transition values	$\delta : \{running, error\}$ (24)
Successor states	$S = \{Running, Terminating\}$ (25)
Description	After receiving the Secure Set, the SSP-Extension builds the Secure Tab on the client-side, which includes all steps of enforcing the Secure Set.

The SSP-Extension starts the *Building* phase by decrypting the received web packet from the web service, analyzing the Secure Set to build the Secure Tab. As relevant for all other states of the protocol, the communication partner (in this case the SSP-Extension) checks if any warnings are added to the response. Then, the first two parameters of the Secure Set are enforced:

- **Extensions:** If the parameter is enabled (set to true), the SSP-Extension now disables all custom browser extensions, which are installed and running on the client-browser. If set to false, no changes are done.

- **Plug-Ins:** Similar to the extensions parameter, an enabled *Plug-Ins* parameter (set to true) signalizes the SSP-Extension to deactivate all installed third-party plug-ins.

For both parameters, an additional list of the disabled extensions/plug-ins is saved in the local storage of the SSP-Extension. The lists are used during the **Terminating** state to restore the client-browser behavior as before the Secure Session started. Additionally, the SSP-Extension registers two listeners to control the status of all plug-ins and extensions. This technique is further described in section 3.3.5. Next the *Content Security Policy* rule is evaluated. The *Content Security Policy* is saved within the SSP-Extension to set up a traffic control point described in section 3.3.5.

The Secure Session is executed in a separate browser window. Before opening the new window, the user is showed a dialogue message which signalizes that the Secure Session is ready and the Secure Tab can be opened. The next parameter of the Secure Set is enforced in this step. The *Private Browsing* parameter influences the newly created window. If the parameter is set to *true*, the new window is created in private browsing mode. The parameter activates the implemented mechanism of all current browsers.

After the Secure Tab is viewed to the user, the Secure Session Protocol continues its execution by entering the **Running** state. The **Establishing** state is successfully executed by the SSP-Extension and the web service. Therefore, a Secure Session, with all security policies enforced, is created and ready to be executed.

3.3.5 Running

Building	
Predecessor states	$P = \{Building\}$ (26)
Transition values	$\delta : \{running, terminating, error\}$ (27)
Successor states	$S = \{Running, Terminating\}$ (28)
Description	The SSP-Extension enters a monitoring mode, controlling all enabled security mechanism while the Secure Session is active.

During the **Running** state of the Secure Session Protocol, the SSP-Extension switches into a monitoring mode after enabling additional security parameters. The SSP-Extension controls all security mechanisms and reports possible violations to the web service. Before switching into the monitoring mode, the SSP-Extension has to set up additional mechanism for the Secure Session.

In order to ensure that all third-party plug-ins and custom browser extension are kept disabled during the Secure Session, the SSP-Extension activates two listeners, which control the installation and manual activation of any plug-in or custom browser extension. The listener is only active, if the respective parameter was previously set to true. If for example, the user wants to install a plug-in or a custom browser extension during the active Secure Session, this action is denied by the SSP-Extension and reported to the web service.

Similar to the listeners, which are controlling belated installed applications at the client-browser, a **traffic control point** is set up to control the proper execution of the Content Security Policy. This policy on the one hand controls the loading of additional resources from locations during the execution of the web service and on the other hand (when not explicitly allowed by the policy [1]) controls the execution of script commands. If a suitable rule matches the request, the client-browser is allowed to load the resource. Otherwise, the request is denied and an error is sent to the web service. Since a violation of the Content Security Policy is a potential security risk (see chapter 4), the security violation is reported to the web service. The Content Security Policy is part of the HTTP Header response by the web service. Before the browser is able to enable the received policy, the SSP-Extension compares the Content Security Policy with the one received encrypted from the web service. The SSP-Extension manually activates the traffic control point by sending the encrypted Content Security Policy directly to the web browser. This action is done to mitigate any kind of manipulation as seen in section 4.

At this point of the protocol all parameters of the Secure Set are enabled and the Secure Session starts with sending out the first web request of the Secure Session. As the destination URL of the first request, the value of the **Entry Point** parameter is used. At this point of the Secure Session Protocol, the Secure Session is established and the SSP-Extension changes its mode to *observing*.

Besides the checks executed by the listener and the traffic control point, the SSP-Extension and the web service have additional tasks to be executed during the Secure Session:

- **Checking the integrity of the web packets:** During the active Secure Session both communication partners add an additional security features to the transferred messages. Before sending a web session packet, the content of the web packet is hashed (e.g. SHA-256) and encrypted with the Secure Session Key. This is done to mitigate an integrity loss during the transfer of the web packet. If the SSP-Extension or the web service determines an integrity loss, this issue is reported and the web service decides if the Secure Session can be continued.

- **Checking the HTTPS public key:** If the web service has specified the parameter *HTTPS Encryption*, the SSP-Extension saves the public key hash delivered within the Secure Set locally at the client-browser. During the active Secure Session, each HTTPS response from the web service is checked whether the public key hash of the HTTPS certificate is valid or not. If the extension determines, that the public key hash of the connection has changed or is invalid, the issue is reported to the web service. The web service upon receiving a warning, decides if the Secure Session can be continued.

- **Reporting Warnings and Errors:** During the Secure Session, it is possible that errors or warnings occur. The communication partners need to handle these messages properly. The SSP-Extension displays the warnings to the user. On the other hand, the web service logs the events and decides if the Secure Session can be continued.

During the active Secure Session, the user is able to browse the web service as usual, until one of three possible events occur, which lead to a termination of the Secure Session. The following listing explains the required conditions, which lead to a transition into the **Terminating** state.

- **Termination by the client-browser:** If the user closes the Secure Tab, the Secure Session immediately terminates. This can be the case, if the user closes the Secure Tab or logs out of the web service.
- **Termination by the web service:** If the SSP-Extension or the web service has determined errors during the Secure Session, the web service can decide to terminate the Secure Session for security reasons. Errors and warnings are displayed to the user.
- **Termination due to a time-out:** If the web service determines that for a fixed time, the client-browser does not send any message through the covert channel, the web service terminates the session. This kind of termination is due to the inactivity of the client-browser. If the user wants to continue, a new Secure Session need to be established.

In the last subsection of the Concept chapter, the **Terminating** state is explained.

3.3.6 Terminating

Terminating	
Predecessor states	$P = \{Ready, Establishing, Pairing, Collecting, Creating, Building, Running\}$ (29)
Transition values	$\delta : \{reset\}$ (30)
Successor states	$S = \{Ready\}$ (31)
Description	The last state of the Secure Session Protocol resets all properties, which were set for the Secure Session and enables the client-browser to execute another Secure Session.

After leaving the Running phase of the protocol the **Terminating** state of the Secure Session Protocol is initiated. In this state of the protocol, the SSP-Extension restores all changes done while building the Secure Tab.

The **Terminating** state of the Secure Session Protocol is reachable from each state of the protocol (except the Ready and the Terminating state itself). If the **Terminating** state is reached, the protocol was possibly terminated due to a raised error. If this is the case, the SSP-Extension and the web service are logging this issue and displaying the user an appropriate error message, which helps the user to improve the security on the client-side.

If the *Private Browsing* parameter is set to the value true, the Secure Tab was initially build with the browser specific option for *Private Browsing*. Therefore, the client-browser itself takes care about removing all created client-data. As soon as the Secure Tab is closed, all Secure Session related data is deleted from the client-side.

During the **Building** phase of the **Establishing** state, the SSP-Extension has disabled all browser plug-ins and custom browser extensions (if the respective rule was set in the Secure Set to the value true). These two actions are reverted in the **Terminating** state. Therefore, the original state of the browser, in terms of active and running plug-ins and extensions, as it was before the Secure Session, is restored.

After all actions are reverted the Secure Session has terminated successfully and the client-browser and the SSP-Extension are ready to establish a new Secure Session. The Secure Session Protocol switches its state back to the **Ready** state, waiting for new outgoing requests.

4 Security Model

In this chapter the **Security Model** of the Secure Session Protocol is explained. The Security Model describes security related properties, which can be achieved if all conditions are met and the Secure Session Protocol is successfully executed. Therefore, this chapter starts by explaining the different security mechanisms, which are introduced through the Secure Session Protocol. After explaining all concepts the chapter concludes by testing the Security Model in regards to the most critical web application security flaws, the OWASP Top 10 [45] in section 4.2.

As explained in the introductory section of this work, several web security mechanisms to enable an appropriate client-side security level exist. Although the web service has been properly used and all standardized web security techniques are in place, the web service has no possibility to check, if the security rules are applied to the client.

The explained Secure Session Protocol in chapter 3 enables the web service to enforce security policies on the client-side. How this is achieved and which other security concepts are important is explained throughout this chapter.

Preconditions

In order classify the achieved Security Model of the Secure Session Protocol, preconditions for the following security concepts are given. The Secure Session Protocol is executed by the web service and the SSP-Extension at the client browser. Therefore, the web service and the SSP-Extension need to be trusted. Since the Secure Session Protocol does not improve the security of any web server, the web service needs to be trustworthy by definition.

To execute the Secure Session Protocol, the SSP-Extension needs to be installed at the client-browser. Hence, two preconditions have to be met. First, the base installation of the client-browser need to be correct and reliable. The base installation is defined as the client-browser without any additional installed applications (e.g. plug-ins or extension). Second, the installed SSP-Extension, as described in chapter 3, need to be trustworthy. The current implementation of the Secure Session Protocol includes an additional security feature, which provides a correct installation of the SSP-Extension (explained in chapter 6) [3].

The following section of this chapter is divided into two subsections. The first part explains different concepts of the Security Model. The subsection starts with an explanation of the security concept introduced through the covert channel between the SSP-Extension and the web service. The subsection continues with a description of the Secure Session and the specified rules and parameters by the web service. The subsection concludes with an explanation of additional security concepts, which further improve the Security Model throughout the Secure Session Protocol. In the second part, the described Security Model is compared with top web security risk, to evaluate the achieved security level.

4.1 Security Concepts

To achieve the desired level of security, the Secure Session Protocol uses cryptographic tools to establish an encrypted channel between the SSP-Extension and the web service. All information transferred through this channel can only be accessed by the communication partners sharing the same Secure Session Key.

4.1.1 Concept of the Secure Session Key

To exchange encrypted messages between the SSP-Extension and the web service, both communication partners need to share a key to establish the covert channel. As described in the section 3.3.2, the Secure Session Protocol uses the Diffie-Hellman Key Agreement Protocol (DH protocol) [42] to create and exchange a symmetric key, which is then used as the Secure Session Key.

The security of the encrypted channel depends on the executed DH protocol. To correctly execute the DH protocol the following conditions have to be met:

The **choice of the protocol parameters** significantly influences the security of the DH protocol. The *generator* g and the *prime group* p need to be selected correctly. The first parameter p, should be chosen by the formula $p = 2q + 1$ where q is a *Sophie Germain Prime* [36]. If p is calculated according to the formula, than p is called a *safe prime* [32]. Further notes how to choose the Diffie-Hellman Key Agreement parameters correctly are given in RFC 2412 on page 45 [42].

As there is always the possibility of a **Man-In-The-Middle Attack**, the SSP-Extension needs to verify the calculation of the web service. Therefore, the additionally exchanged Out-of-Band secret is used to verify the correct origin of the calculation.

The exchanged Secure Session Key can be securely used for encrypting information, which is exchanged between the web service and the SSP-Extension. All parameters used for calculating the Secure Session Key are deleted afterwards. Therefore, the DH protocol trivially achieves the property of Perfect Forward Secrecy [37].

[3] This technique is obsolete for further developments of the Secure Session Protocol as seen in section 6.4

HTTPS and Mutual Identification

As described in the introductory chapter of this work, the Secure Session Protocol should be used by web services, which are working with sensitive data. These web services should always use the HTTPS Protocol to encrypt the traffic between the web service and the client-browser. The attentive reader might wonder, why this encrypted connection is not used as a covert channel for exchanging data between the web service and the SSP-Extension.

The HTTPS Connection encrypts the connection between the browser and the web service. In contrast to that, the Secure Session Key encrypts the connection between the SSP-Extension and the web service. The difference between both channels is the access to the decrypted data. For the case of the HTTPS connection, the client-browser is able to read and modify the decrypted data. On the other hand, data sent through the covert channel is only accessible for the SSP-Extension and the web service. Furthermore, the SSP-Extension has access to both communication channels, as the SSP-Extension is able to read and modify the HTTPS connection. In figure 13 the difference between both channels is visualized.

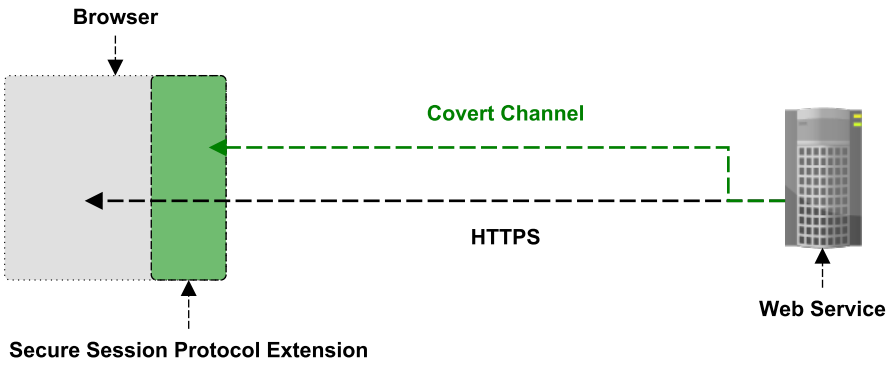

Figure 13: Difference between the covert channel and the HTTPS connection

Additionally, the Secure Session Key is used as an **identifier** for the client-browser. The key is unique for each pair of user-ID, client-browser and web service, because a Secure Session without the Secure Session Key (which is securely stored) is not possible. Therefore, after the Secure Session Key is exchanged, the web service can clearly identify the user. By now, it was only possible for the user to identify the web service through checking the HTTPS certificate. By introducing the concept of the Secure Session Key, the Secure Session Protocol enables **Mutual Identification**.

To understand why the just explained channel is important for the achieved security the role and the different tasks of the SSP-Extension are explained in the following paragraph:

Role of the SSP-Extension

Current web services have no possibility to receive (or collect) correct information about the client-browser, i.e., the web services would not detect any changes by a third party. Currently, the client-browser adds HTTP header fields to the request, which are containing the required information. Since this additional provided information could be altered by the user or manipulated by an adversary, the web service cannot trust the delivered information. By using the covert channel, this problem is solved and therefore enables the security mechanism **Receiving Correct information**.

The SSP-Extension is described as a trustworthy partner of the web service at the client-browser. The SSP-Extension has the task of executing the Secure Session Protocol on the client-side. Furthermore, the SSP-Extension executes task for the web service, which it is not able to execute alone. Therefore, the security mechanism **Enforcing Client-Side Policies** is created. The tasks, executed by the SSP-Extension for the web service, are described in the following listing:

- **Collecting Information:** As described in section 3.3.3, the web service needs correct information about the client-browser to be able to create a proper Secure Set for the client-browser. Correct information are unavailable to the web service due to two reasons. First, parts of the collected data (described in section 3.3.3) are currently not available to the web service. For example, the status and version of installed third-party plug-ins is currently not sent to the web service. Second, all information sent by the client-browser cannot be trusted, as an adversary (or a third-party) could potentially have modified transmitted web packets. Therefore, the SSP-Extension collects all relevant information for the web service.

- **Controlling the network traffic:** Looking at the transferred packets between the web service and the client-browser, without the SSP-Extension, the web service alone is unable to check whether or not the web traffic was modified during the transfer over the network. By using the encrypted channel between the SSP-Extension and the web service, the protocol is able to check the integrity of the transferred web packets at both ends of the connection.

- **Monitoring the client browser:** The SSP-Extension has fully access to the client-browser as it is part of the client-browser. [4]. This enables the SSP-Extension to enforce and monitor all provided security rules by the web service.

The combination of the SSP-Extension as a trustworthy partner on the client-side, together with the covert channel between the SSP-Extension and the web service, enables the web service to clearly **identify** the user and **enforce** security parameters on the client-side.

Furthermore, due to the fact that the web service is able to specify the Secure Set for each Secure Session, the security can be further improved. By default (without the Secure Session Protocol) a web service needs to **statically** specify the security parameters and rules for a requesting user. Shown in the previous paragraph, the web service has no possibility to retrieve any reliable information about the client-browser. By using the Secure Session Protocol, the web service can create the security rules **dynamically** by adapting them to the correct information collected by the SSP-Extension. Thus, by individually creating the Secure Set for each Secure Session, the security level is further increased.

4.1.2 Security Model of the Secure Session

The following subsection gives an overview of the Security Model of the Secure Session. The security level of the Secure Session significantly depends on the rules given by the web service. Hence, this section describes the possible security level that can be achieved by setting the parameters and rules. These parameters are need to be individually set for each client-browser, thus only general advices are given in this subsection. The following listing explains each parameter and rule of the Secure Set. The mitigated attacks are written bold:

- **Private Browsing:** The Private Browsing mode of the Secure Session Protocol enables *the built-in Private Browsing mode* within the Secure Tab. This parameter of the Secure Session Protocol increases the privacy of the user. All session (in this specific case the Secure Session) related information are deleted afterwards [26], as no sensitive or session related data is stored on the computer. Thus, making web security attacks such as **reusing opened sessions** or the **exposure of sensitive data** hard.

- **Manage additional browser Add-Ons:** The base installation of current browsers is able to execute nearly every today's web service. To enhance the user experience, browser vendors such as Google or Mozilla introduce the concept of custom browser extensions. These custom browser extensions let the user personalize their browser. A custom browser extension is therefore not mandatory for any web service. In contrast to the custom browser extensions, third-party plug-ins, such as the Adobe Flash Player [2] or the Java Runtime Environment [47], are required for finite web services. Both types of browser add-ons have drawbacks in regards to the security of the client-browser.

  Custom Browser Extension: *(this subitem refers to Google Chrome's custom browser extensions)* The permissions of custom browser extensions are equally treated as current Android Mobile OS Applications [14] [25]. The *manifest.xml* file specifies the different permissions of the application, which are displayed to the user during the installation. Likewise in the case of Android mobile application, the average user is unable to correctly interpret the permissions or even worse, does not read the different permissions [11]. Therefore, the installation of a malicious extensions is easy and hard to detect afterwards. As custom browser extension are able (depending on their given permissions) to execute different kinds of web security attacks, not trusted custom browser extensions can reduce the security level of the client-browser. By deactivating custom browser extension throughout the Secure Session a wide range of web security attacks can be omitted. For example, the **exposure of sensitive data**, **unvalidated Redirects and Forwards** or **Compromising the Network Integrity**.

  Third-party Plug-ins: Like custom browser extensions, third-party plug-ins can be exploited for web security attacks. Different to custom browser extension, installed plug-ins have the permission to access the underlying operating system. Thus, a successful attack has a bigger impact on the security. As plug-ins are usually delivered

[4] Full control is not correct for the current implementation. The SSP-Extension has no possibility to control browser specific options and the operating system. However a solution is given in section 6.4

by trusted third-parties, the executed software is not malicious. Nevertheless, plug-ins could have potential implementation errors and are therefore an entry point for an adversary to execute the web security attack. Such errors can be fixed through updates. We consider that up-to-date software components cannot be exploited by an adversary. Due to the fact, that the average browsers are not up-to-date, such errors are sustaining [13]. By equally managing plug-ins, such as custom browser extensions, outdated plug-ins can be detected and the user can be informed about a potential security issue. Deactivating plug-ins during the Secure Session increases the security level of the Secure Session. As mentioned above, deactivating outdated software components mitigates attacks such as the danger of **Using Known Vulnerable Components**.

- **Entry Point Correction:** To mitigate **CSRF** attacks, the web service is able to specify the correct Entry Point of the web service. During the Secure Session **CSRF** attacks are not possible. Further, *correcting the Entry Point* enables the web service to enforce HTTPS on the client-side.

  Enabling HTTPS: To further strengthen the security of the Secure Session, the web service can check with the help of the SSP-Extension, which type of web protocol (HTTP or HTTPS) is currently used. By changing the Entry Point of the web request, the web service can specify that the HTTPS connection should be enabled (by changing the protocol to HTTPS). Furthermore, the web service can set the HTTPS encryption parameter value to the public key hash of the web service. This enables the SSP-Extension to deny **Man-In-The-Middle** attacks against the HTTPS connection. A possible attack, including a **Man-In-The-Middle** attacker who changes the HTTPS certificate, is called **SSL-Stripping** and was introduced in [39].

- **Content Security Policy:** The Content Security Policy is the only security rule, the web service is able to specify for the Secure Session. The Content Security Policy, which was developed by the Mozilla Foundation [1], specifies a white-list of allowed locations for loading additional resources. The policy describes allowed locations from which a requested resource can be loaded. Further, by default the in-line execution of script resources (JavaScript) and the evaluation of arguments (using the function eval()) are forbidden. As described in the original specification of Mozilla Foundation, the Content Security Policy protects against **Cross-Site-Scripting** [1]. In addition, the Content Security Policy is able to mitigate **Clickjacking** and **Packet Sniffing** attacks.

  Extending the Content Security Policy: To mitigate another class of web security attacks, the Content Security Policy is improved such that all outgoing requests are checked. The standard Content Security Policy only checks, if the URL of the requested resource is matched within the white-list. Normally, outgoing web requests are not monitored. For standard web services this is a desired feature (enables the user to change the URL). For the case of the Secure Session, the user only visits the specified web service within the Secure Tab. Therefore, the extended Content Security Policy checks all outgoing web requests of the Secure Tab. This mitigates additionally all kinds of **Phishing** and **Unvalidated Redirects and Forwards** attacks

4.1.3 Additional Features

Besides the Security Concepts enforced by the Secure Session, the Secure Session Protocol further increases the security during the runtime of the protocol with the following concepts:

- **Timeout:** As introduced in chapter 3, the web service and the SSP-Extension monitor the behavior of the user in terms of inactivity. If one of the communication partners determines an inactivity, the connection is terminated due to a *timeout*. If the user wants to continue, the Secure Session needs to be re-established. Therefore, any kind of **Active Session** attacks can be mitigated.

- **Network Integrity:** Attached to the encrypted packet transferred between the web service and the SSP-Extension, the communication partners are adding a checksum of the current web packet. This enables the receiver, to check the integrity of the web packet and thus the reliability of the network. As the checksum is encrypted with the Secure Session Key, only the web service and the SSP-Extension is able to verify the integrity of the transmitted web packet. This type of advanced security mechanisms mitigates any kind of web security attacks involving the missing integrity of the network, such as **Man-In-The-Middle** attacks.

- **Client-Side Modifications:** Mentioned in chapter 3 and described at the beginning of this chapter 4, the SSP-Extension is a trustworthy partner of the web service at the client-side. The SSP-Extension is able to extract correct information about the client-browser and the client-side environment. Some information collected during the **Collecting** phase of the Secure Session Protocol are redundant (sent by the client-browser and added by the SSP-Extension). This might be a drawback in regards to performance and overhead, but it further improves the security of the Secure Session Protocol. As the client-browser is unable to modify the information sent by the SSP-Extension, the web service can determine if the client-browser sends false information, e.g., a modified user-agent.

The web service is therefore together with the SSP-Extension able to determine **Client-Side Modifications** of the web packets.

4.2 Evaluating the Security Model

The chapter concludes, by comparing the Secure Session Protocol Security Model with the OWASP Top 10 [45]. The listing shows, which web security attacks can be mitigated through the Secure Session Protocol.

Note: *As A1, A4, A5 and A7 of the OWASP Top 10 are web service specific, the listing only contains the relevant client-side attacks.*

- **A2 Broken Authentication and Session Management** - The top client-side security risk of the OWASP Top 10 List includes all attacks related to a compromised authentication or web session. Those type of web security attacks (client-side) can be mitigated through the Secure Session Protocol. Responsible therefore are different security concepts like the **mutual identification** by the Secure Session Key, the **Timeout** mechanism or **HTTPS enforcement**.

- **A3 Cross-Site Scripting (XSS)** - Similar to *Injection* attacks described at A1, XSS attacks are describing all attacks, which occur when untrusted input data is evaluated at the client-browser. An attacker tries to inject data which is interpreted as script code (e.g. JavaScript). Such attacks are completely solved through the **Content Security Policy**. As this policy is definitely enforced by the Secure Session Protocol, all XSS related attacks are solved.

- **A6 Sensitive Data Exposure** - Number six of the top web security risk of 2013 describes all attacks, which are able to expose sensitive data. Data exposure can occur due to several reasons. First, it can be due to no or insecure cryptographic usage for encrypting the network traffic. Second, additional software components can expose data, because they are malicious or have security vulnerabilities, which are exploited by an adversary. Another possibility, are faulty web service components such as advertisement frames or media resources, which try to expose sensitive data by extracting information from the client-browser. The Secure Session Protocol minimizes this attack vector by introducing the different security concepts such as **HTTPS enforcement** or the **extended Content Security Policy**.

- **A8 Cross-Site Request Forgery (CSRF)** - Besides XSS, Cross-Site Request Forgery (CSRF) is one of the top web security risks on the client-side. An attacker tries to use the authenticated user (authenticated against the web service) to execute commands for him. In contrast to the user, the adversary is not privileged to execute commands at the web service. Therefore, e.g., manipulated URI's are used, which are tricking the user to execute commands for the attacker. Due to the **mutual identification** mechanism of the Secure Session Protocol, only the intended client-browser user is able to execute web service related commands.

- **A9 Using Known Vulnerable Components** - The ninth top web security risk of the OWASP Top 10 describes attacks which are arising due to security issues from vulnerable software components on the web server side and as well on the client-side. A vulnerable component is a software product or part of a software product, which has a security vulnerability in a specific version of the software. This vulnerability can be exploited by an adversary to start an attack. The Secure Session Protocol stops these kind of web security attacks by controlling all installed software components at the client-browser. This includes all installed custom browser extensions, installed third-party plug-ins and as well the client-browser version. Described in chapter 6.4 the Secure Session Protocol implementation can be extended to further control other software products installed on the client-computer.

- **A10 Unvalidated Redirects and Forwards** - Due to hyperlink elements or script based location changes, the client-browser gets redirected to other pages. Those redirect can depend on untrusted input. Therefore, an attacker can use this technique to trick the user into so called *Phishing* web pages. To mitigate all kinds of **Unvalidated Redirects and Forwards** the Secure Session Protocol extends the Content Security Policy to forbid any kind of redirect at web pages, which are not explicitly specified by the web service.

5 Example

After the explanation of the Security Model, in this chapter an example execution of the Secure Session Protocol is presented. The client-browser establishes a Secure Session with a Type III web service (see 1.1). The example explains all relevant concepts described in chapter 3. Additionally, details about how the mentioned security features and mechanisms in chapter 4 are enforced are given.

The following example is divided into smaller paragraphs. Each of them describes a single state of the Secure Session Protocol. Furthermore, at the beginning of each paragraph a figure sums up all relevant information transferred between the communication partners during the execution of the state. Before continuing with the description of the example, necessary assumptions are made. Therefore, details about the web service and the client-browser are given.

Assumption

The SSP-Extension is successfully installed at the client-browser and the web service is ready to establish a Secure Session. Both communication partners are therefore able to execute the Secure Session Protocol.

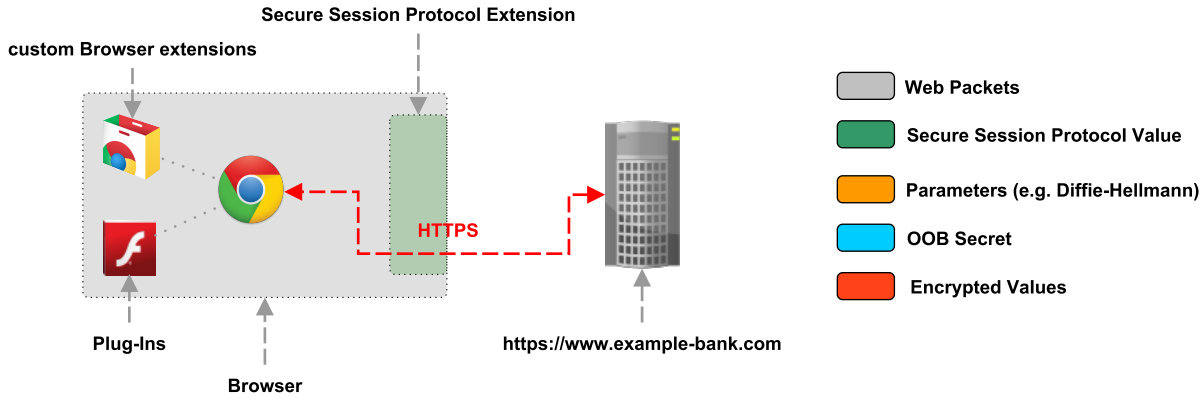

Figure 14: Overview of the example participants including a legend for the different used data types

The requested web service in this example is a Type III web service. By definition this type of web service has only a private section and processes sensitive data (see section 1.1 for more details). The used protocol for transferring web packets is the HTTPS protocol. The URL of the web service is `https://www.example-bank.com/`. The web service is an online banking web service, which is used for managing and processing *online banking transactions*. The web services uses a *Flash-Animation* (requires the third-party plug-in Flash Player) on the web page to randomly generate small image patterns. These images are used, together with a (physical) TAN generator on the client-side, to generate TAN's over an Out-of-Band channel. This mechanism is used by several online banking services to securely confirm the transaction. An explanation of this technique can be found on [30].

The installed client-browser is up-to-date and the base installation is trustworthy. Additionally, custom browser extensions are installed and the client-browser supports different third-party plug-ins. The Flash-Animation plug-in, required from the online banking web service, is already installed, but in a recent outdated version. Hence, the installed version is still secure.

In figure 14 the preconditions are visualized for the reader. Additionally, a color legend to identify the different web packets is given. The SSP-Extension starts monitoring the traffic from the client-browser, waiting for a web service able to communicate through the Secure Session Protocol. Therefore, the protocol starts in the **Ready** state of the Secure Session Protocol.

Ready

The Secure Session Protocol is initialized right after the start of the client-browser. The SSP-Extension is active and enters the **Ready** state. The SSP-extension monitors all outgoing web requests to be able to start initiating a Secure Session. The user opens the online banking web service by pressing a URL link from the *bookmark tool bar*. The following URL is requested: `https://www.examplebank.com/showSales`. The bookmark was set by the user in the past, as he was looking at his transactions. The SSP-Extension intercepts the web request, by adding the Secure Session Protocol value

initialization (see figure 15).

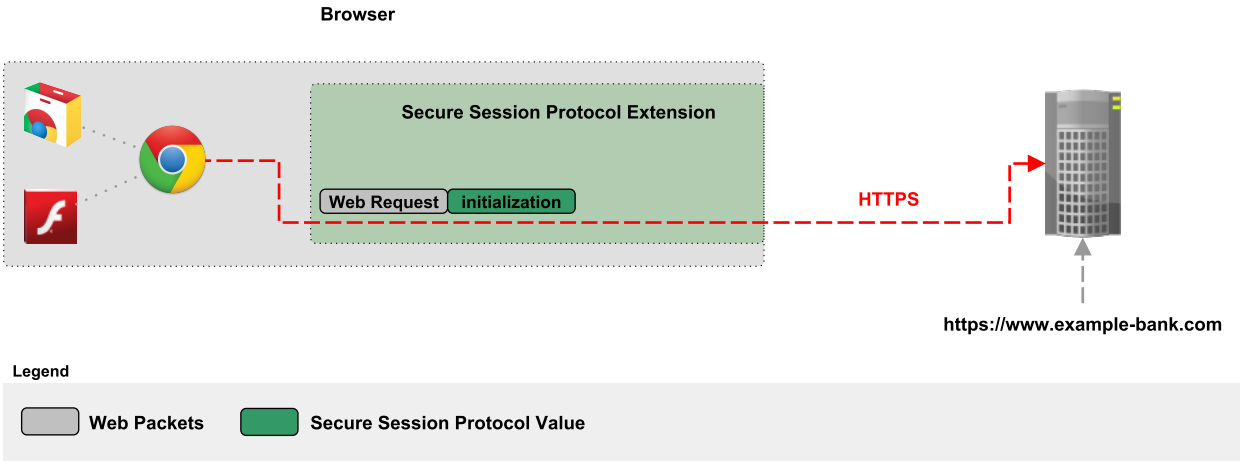

Figure 15: Secure Session Protocol starts the protocol by sending out the Secure Session Protocol Value

The web service receives the request by the user, recognizing the Secure Session Protocol value. Because the web service is able to execute the Secure Session Protocol, it initiates the **Pairing** state of the protocol.

The web service starts the **Pairing** state, as described in section 3.3.2, by adding the first part of the Diffie-Hellman Key Agreement to the response, as well as the Secure Session Protocol value *pairing* to indicate the SSP-Extension that the web service is able to execute the Secure Session Protocol and wants to start the pairing phase.

Pairing

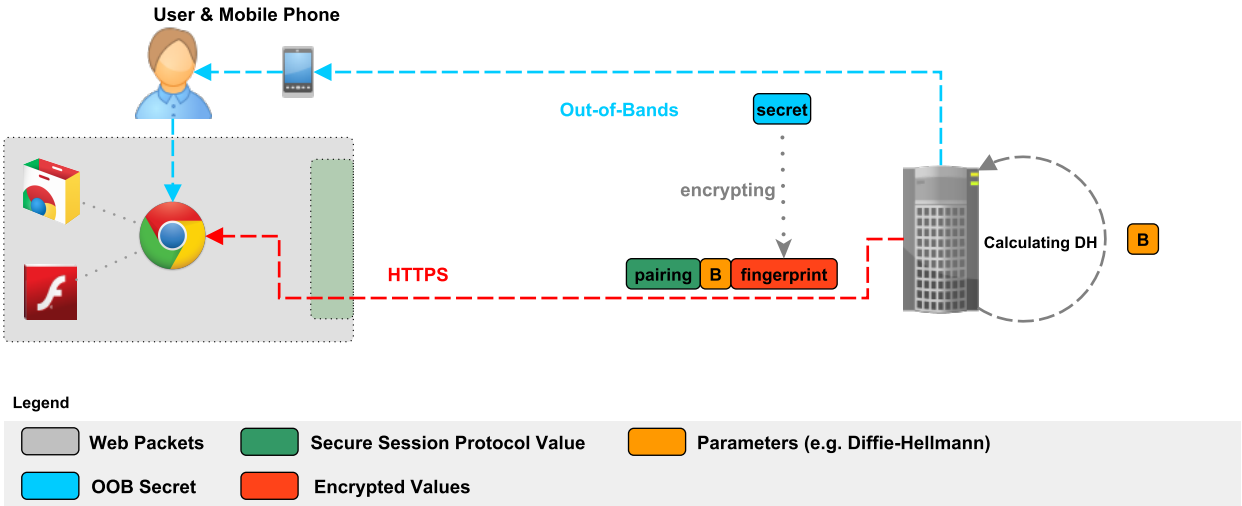

Figure 16: The web service calculates the parameters of the Diffie-Hellmann Key Agreement

In figure 16 all activities related to the response of the web service during the start of the pairing protocol are visualized. To enable the SSP-Extension to verify the first part of the Diffie-Hellman Key Agreement, the web service needs to transmit an additional Out-of-Band secret to the SSP-Extension. In this example, the web service uses the mobile phone of the user to transmit the Out-of-Band secret (the mobile number of the user was previously exchanged). As the channel from the web service to the mobile phone of the user uses the cellular network, the channel is a valid Out-of-Band channel. In [54] the technique of sending an Out-of-Band secret to verify transactions is presented and the achieved security is described. The web service encrypts the public key fingerprint of the HTTPS certificate with the send Out-of-Band

secret and adds the cipher to the response.

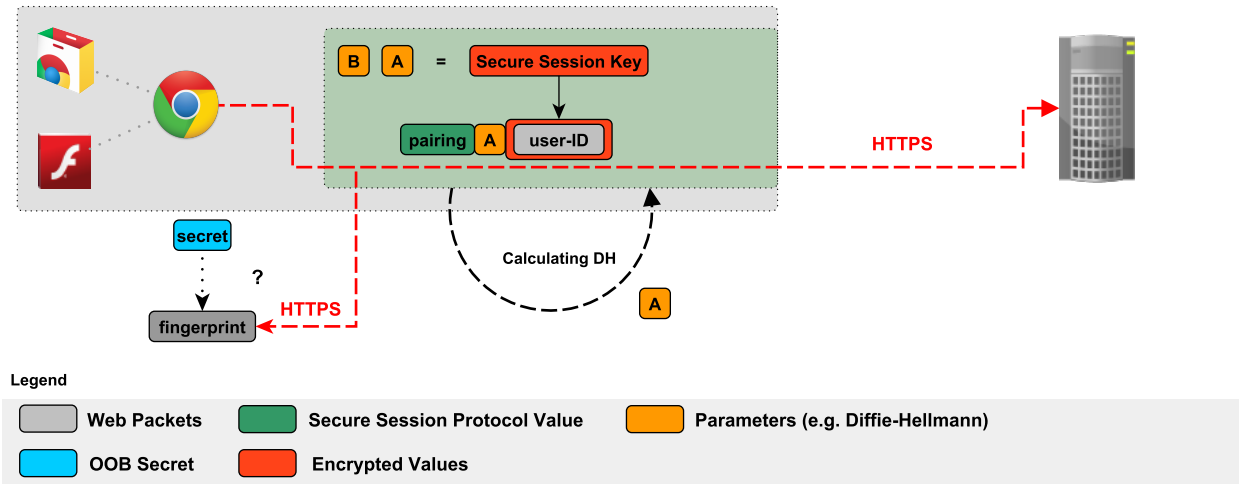

Figure 17: The SSP-Extension calculates the Secure Session Key

The SSP-Extension receives the first part of the Diffie-Hellman Key Agreement B and over the Out-of-Band channel, the additional $secret$ (for the verification of the identity of the web service) is received. The SSP-Extension verifies the first part of the Diffie-Hellman Key Agreement, by decrypting the fingerprint f of the web service with the Out-of-Band secret. The SSP-Extension compares it to the extracted fingerprint from the HTTPS certificate f'. The check is successful and the SSP-Extension continues with calculating the second part of Diffie-Hellman Key Agreement (A) and the Secure Session Key. As seen in figure 17, the SSP-Extension calculates the Diffie Hellmann parameter A for the web service, which enables the web service to also calculate the Secure Session Key.

The SSP-Extension sets up a response for the web service, which includes the second part of the Diffie-Hellman Key Agreement, the Secure Session Protocol value *pairing* and an encrypted packet including the user-ID of the user. The last part of the packet is used for a verification of the functionality of the Secure Session Key on the web service side. If the web service is able to create the correct Secure Session Key, it is also able to decrypt the packet and can read the correct user-ID. In figure 17, the complete web packet transmitted to the web service is shown.

After receiving the web packet from the SSP-Extension, the web service starts calculating the Secure Session Key appropriate to the Diffie Hellmann Key Agreement specification. To verify that no adversary has modified the parameters of the key exchange, it checks the user-ID encrypted with the Secure Session Key. Since, the check was successful, the communication partners are sharing the same Secure Session Key. Hence, are now able to establish a Secure Session on the client-side.

The protocol switches into the **Establishing** state of the Secure Session Protocol as both are sharing the same pairing information. The web service indicates this by sending the Secure Session Protocol value *establishing* encrypted to the SSP-Extension.

Establishing

The SSP-Extension continues the protocol by entering the **Collecting** phase as it is shown in figure 18. As defined in section 3.3.3, the SSP-Extension extracts the following correct information about the client-browser for the web service:

- **Host (requested URL):** `https://www.example-bank.com/showSales`
- **User Agent:** Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.131 Safari/537.36
- **Custom Browser Extensions:** 5
- **Plug-ins:** [Adobe Flash Player, 13.0.0.206], [Silverlight, 5.1.30214.0], [Java(TM), 10.55.2.14]
- **Network status:** 0

The *Extensions* parameter symbolizes that in total five custom browser extensions are currently installed at the client-browser. For the later achieved security, it does not matter what the name and the version of the custom browser extensions are. All listed information are packed together and encrypted with the Secure Session Key. Added to this web packet are also the user-ID for the web service and the Secure Session Protocol value *Collecting*. The complete web packet is sent to the web service, as shown in 18.

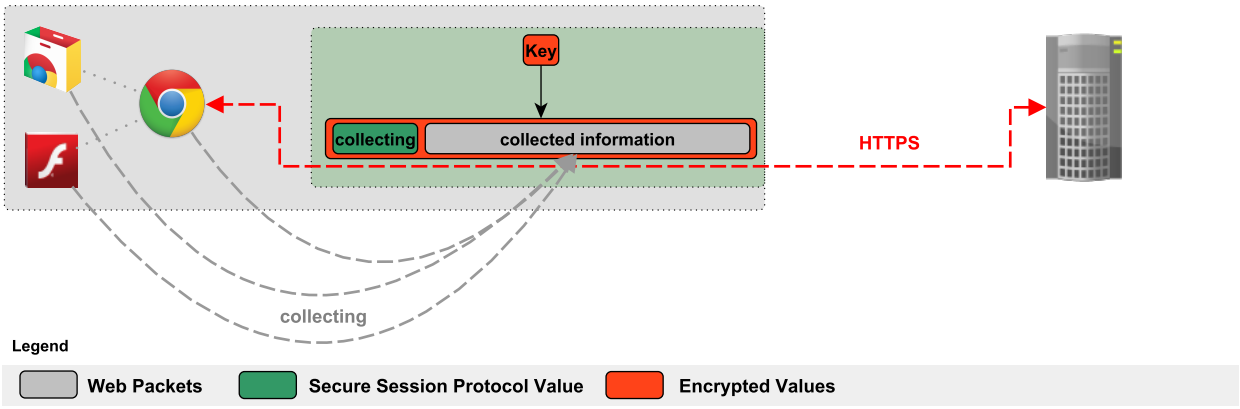

Figure 18: Collecting authentic information for the the Web service

The web service is able to decrypt the received list of collected information by using the Secure Session Key. Depending on the information collected by the SSP-Extension the web services continues the execution by doing initial security checks and afterwards building the Secure Set.

The web service continues by checking the browser version, browser vendor and the installed version of the Flash Player plug-in (these three tests are the initial security checks defined by the specific web service). The Flash Player plug-in, as described in the assumptions paragraph of the example at the beginning of this chapter, is required for generating a TAN for a secure payment. The plug-in version is important and therefore checked by the web service. In the following figures 19 and 20, are showing examples for initial security checks by the web service:

```
if (userAgent.browserVendor is "Chrome" && userAgent.browserVersion >= 30.0.0)
    return success
else if (userAgent.browserVendor is "Firefox" && userAgent.browserVersion >= 5.0.0)
    return success
else if (userAgent.browserVendor is "Internet Explorer" && userAgent.browserVersion >= 8.0.0)
    return success
else
    return error
```

Figure 19: Pseudo code for checking the requesting user-agent

The client-browser is in this case the Google Chrome browser with the version *34.0.1874.131*. The policy of the web service is fulfilled (see pseudo code example figure 19). The second security check evaluates the Flash Player plug-in version.

The installed plug-in version of the third-party Flash Player is *13.0.0.206*. Accordingly to the policy shown in 20, the Secure Session Protocol has found a potential security vulnerability and therefore a warning is raised. The warning notifies the user that the version of the plug-in is outdated, but is yet valid and the Secure Session for this web service can be continued. A warning is added to the response for the SSP-Extension by the web service. It includes a message which tells the user that the version of the Flash Player plug-in is not up-to-date and an update is recommended due to security reasons.

```
if ( plugins.FlashPlayer.Version >= 14.0.0 )
    return success
else if ( 14.0.0 >= plugins.FlashPlayer.Version >= 13.0.0 )
    return warning
else
    return error
```

Figure 20: Evaluating the Flash Player version against the security policy by the web service.

Before sending out the warning to the SSP-Extension, the web service starts the *Creating* phase by setting up the Secure Set for client-browser. The following listing shows how the parameters of the Secure Set are set by the web service:

- **Private Browsing:** The web service does not want to save any, possible sensitive, session data on the client-side. The parameter *Private Browsing* is set to *true*. Any created session data is deleted after the Secure Session is terminated.
- **Extension:** The collected information from the SSP-Extension specifies that the client-browser has active custom browser extensions installed within the client-browser. To avoid potential security issues as described in chapter 4, the parameter *Extensions* is set to the value *true*.

  Note: *If no custom browser extensions are installed before starting the Secure Session it can be useful to set this parameter nevertheless to true. This enables the SSP-Extension to keep track of newly installed custom browser extensions during the execution of the Secure Session.*
- **Plug-ins:** The web service has not found any suspicious third-party plug-ins installed on the client-side. Hence, the Secure Session can be established without deactivating any plug-ins. The parameter *Plug-ins* therefore is set to the value *false*.
- **Entry Point:** For evaluating the parameter *Entry Point*, the web service examines the value of the request URL collected by the SSP-Extension. The requested URL from the user is `https://www.example-bank.com/showSales`. The *Entry Point* parameter is set to `https://www.example-bank.com/` since this is the correct start web page of the web service.
- **HTTPS Encryption:** The web service in this example is a Type III web service which uses the HTTPS protocol for transmitting data. Thus, the Secure Set parameter *HTTPS Encryption* is set to the hash of the public key of the HTTPS certificate.
- **Content Security Policy:** The Content Security Policy for this web service is strict. Any additional resource, like scripts, images or media files needed to be loaded by the web service, can only be loaded from the default host, which in this example is https://www.example-bank.com/. Furthermore, the web service forbids the execution of the function *eval* and the *inline-script* execution of script code. The resulting Content Security Policy is:

Content-Security-Policy : default-src https://www.example-bank.com/;

Figure 21: Content Security Policy of the example web service

The created Secure Set is encrypted together with the Secure Session Protocol value *building* and the raised warning during the security checks. This warning includes a message to the client mentioning, that the current version of the installed Flash Player plug-in is outdated and could be a potential security risk in the future. The creation of the Secure Set as well as the transmitted packet is shown in figure 22.

The SSP-Extension continues the execution of the protocol by building the Secure Tab on the client-side. After decrypting the received packet from the web service, the extension starts creating the Secure Tab. First, the added warning is processed. The warning is displayed to the user showing the message about the outdated plug-in version.

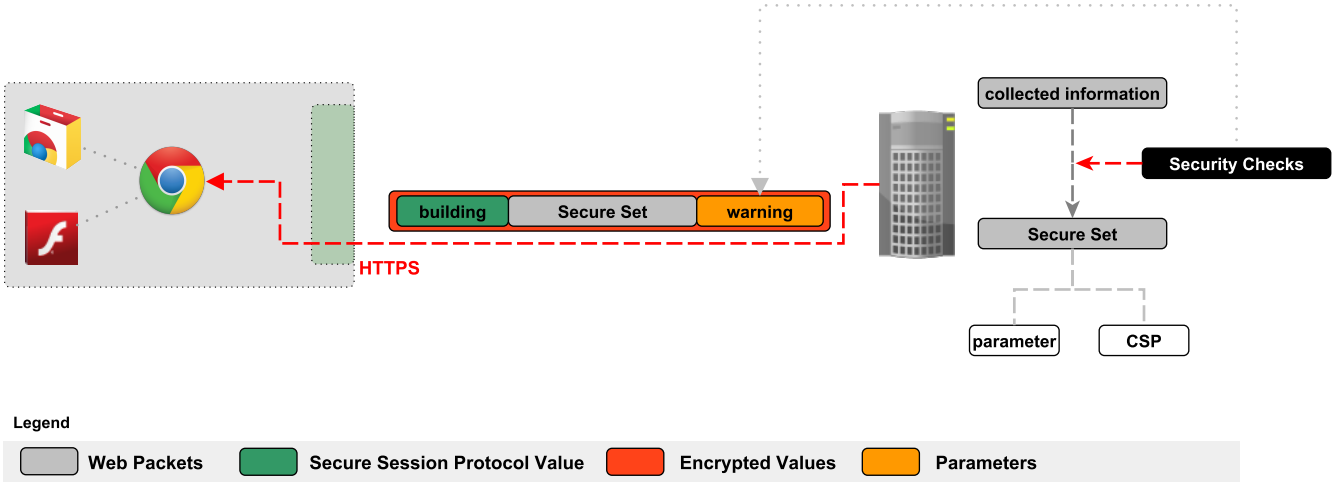

Figure 22: Based on the received information the web service creates the Secure Set

The SSP-Extension continues with building the Secure Tab. As defined by the Secure Set, the browser window will be in *Private Browsing* mode. Before the Secure Session starts, the SSP-Extension disables all custom browser extensions and sets up the traffic control point by directly setting the Content Security Policy at the client-browser. The plug-ins are kept unchanged, because the web service has set the parameter to false.

Running

The Secure Session starts with the opening of the Secure Tab. The first web request of the Secure Session is set to the *Entry Point* parameter of the Secure Set which is in this example is `https://www.example-bank.com/`. Added to this web request, is the Secure Session Protocol value *running* to indicate the web service that the Secure Session was successfully established on the client-side.

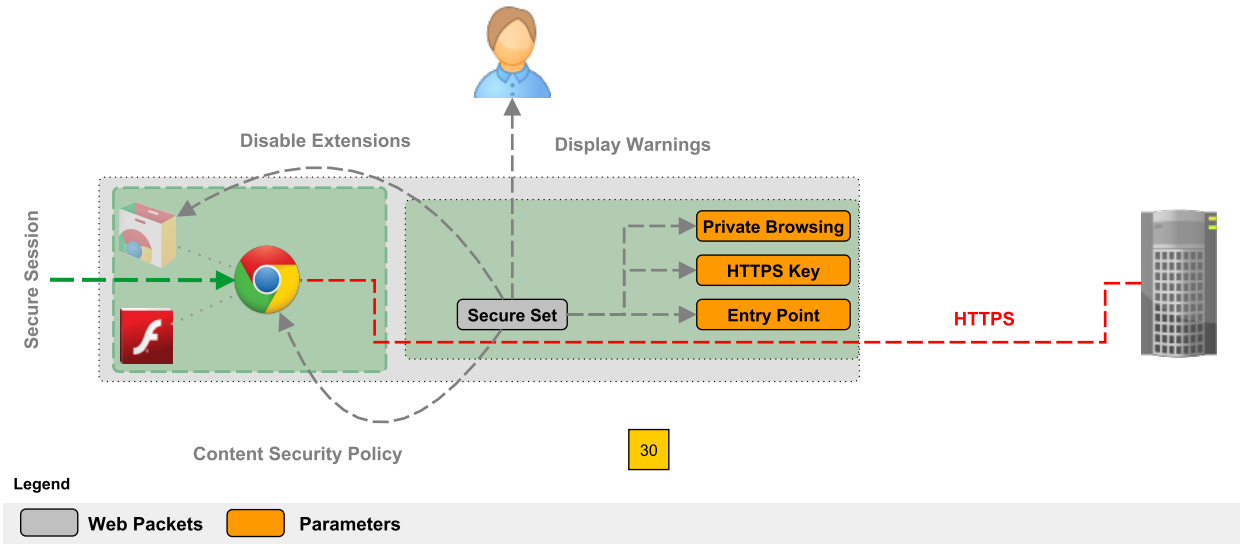

Figure 23: The successfully created Secure Session on the client-side

The Secure Session is now running and the SSP-Extension switches to the observing mode by controlling the listener and the traffic control point (shown in figure 23), until the Secure Session is terminated by either side. Additionally, the SSP-Extension and the web service both add an encrypted checksum to the original transferred packets to enable the opposite communication partner to check the integrity of the web packets. Furthermore, the public key hash of the

HTTPS certificate of the incoming connection is checked.

Terminating

In this example the user on the client-side finishes the Secure Session by terminating the web service. The web service therefore sends the Secure Session Protocol value *terminating* to the SSP-Extension indicating that the Secure Session is terminated.

During the **Terminating** state of the Secure Session Protocol, all changes to the browser have to be restored. Therefore, the SSP-Extension disables the traffic control point and re-activates all custom browser extension. Further, the build-in *Private Browsing* mode deletes all created data of the Secure Session. After all actions have been reverted, the Secure Session Protocol is finished and the protocol switches back to the **Ready** state.

6 Implementation

This chapter shows and explains an (concept-) implementation of the Secure Session Protocol. The implementation described in this work is a prototype and is not a solution that can be used in a production environment. The goal of this chapter is to show how certain aspects of the Secure Session Protocol, as they are described in the concept chapter, can be achieved. In Addition, problems of the current implementation approach and further development steps are described.

The section starts with a general explanation of the implementation of the different protocol participants. The paragraph includes detailed information about the used technologies and components. Then, the implementation of the SSP-Extension is explained. The current implementation of the Secure Session Protocol is based on a *Google Chrome's* browser extension API. Hence, an explanation how the extension executes the different Secure Session Protocol mechanism is given. The implementation chapter concludes with an outlook of existing problems and open tasks towards a productive solution.

The final implementation of this work includes the SSP-Extension and a showcase web service with Secure Session Protocol support. The user is able to execute each Secure Session Protocol state. Each step of the user is explained and information about the protocol is given. Additionally, the user is able to read and modify web service specific parameters of the Secure Session Protocol. Further, hidden information, e.g., Diffie-Hellmann Key Agreement parameters, are shown to the user to increase the understatement of the single states of the protocol.

6.1 Protocol Participants

Described in the concept section of this work, the Secure Session Protocol in total consists out of three participants. In this section, each participant is described in regards to used technologies.

The current implementation of the **web service** is a set of PHP web pages (implemented in version 5.5.11) [49], which are guiding the user through the single states of the Secure Session Protocol. All protocol mechanism are implemented in PHP, by using native and third-party libraries. For establishing and using the covert channel between the web service and the SSP-Extension, the web services uses the slowAES library [52] for de- and encrypting and the phpseclib-BigInteger [50] add-on library for executing the Diffie-Hellmann Key Agreement. Further the phpseclib library is used [51]. In addition to the server-side execution, the web services delivers Bootstrap (version 3.2.0) [66] CSS3, JavaScript and HTML5 on the client-side. Figure 24 shows a picture of the current implementation during the **Establishing** state of the Secure Session Protocol. The web service was solely tested on Google Chrome, because the SSP-Extension is implemented as a Google Chrome custom browser extension. However, as the web service is based on standard web technologies, the web service is runnable on all current browser versions, which are supporting HTML5. To be able to execute the Secure Session Protocol tasks, the web service needs access to the HTTP header fields sent by client-browser. The PHP command *getallheaders()* (supported since PHP version 4.3.0) [48] returns all HTTP headers of the current web request. This command is supported by Apache web servers only. Therefore, the implementation uses the current Apache version 2.4.9 [61].

For testing and using the Secure Session Protocol implementation of this work, Google Chrome [22] by Google Inc [23] is used as the **client-browser**. This is due to the fact, that the current implementation of the protocol is based on Google Chromes custom browser extension API. Hence, the Secure Session Protocol is a temporarily addition to the browser and can be installed and removed by user. Therefore, no modifications to the base installation of the browser were done. The version of the browser used for this work is *35.0.1916.153m*.

6.2 SSP-Extension

The SSP-Extension is implemented as a Google Chrome custom browser extension and uses the Google Chrome JavaScript Platform API [24]. In total, the SSP-extension consists out of two different extension concepts. At first, a JavaScript *backgroundPage* [15] controls the execution of the single steps of the Secure Session Protocol on the client-side. The extension therefore specifies the background page to be persistent (always active). In combination with the *scope* of the background page (defined in the manifest.json file), the JavaScript execution is active for the whole life cycle of the client-browser. The *scope* of the background page thus is set to **`[*://*/*.]`** to enable the extension to interact with **all** incoming and outgoing web requests on every web page. Besides the background page, the SSP-Extension uses a *browserAction* to show the user important information about the current status of the Secure Session Protocol. Further, warnings, errors and termination messages are displayed through the *browserAction* [16] interface.

Besides the permissions, which defines the scope of the background page, several permissions are needed to execute all relevant steps of the Secure Session Protocol. The permissions used by the Secure Session Protocol are listed in

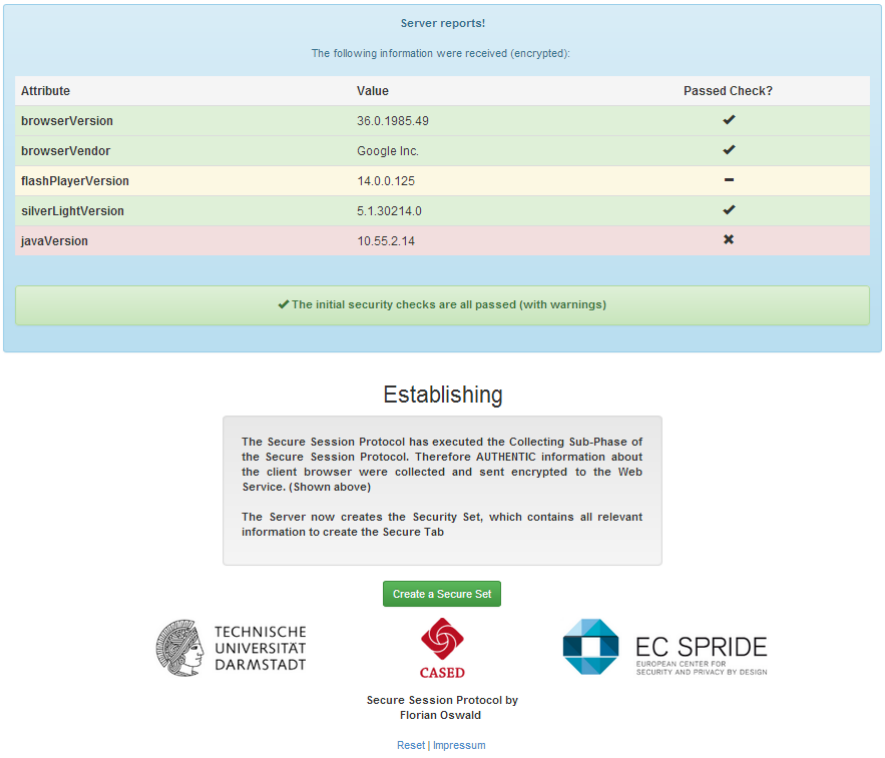

Figure 24: Image of the web server implementation during the Establishing phase

the *manifest.json* file. The following listings shows all permissions and explains for which purpose of the protocol the permission is used for:

- **tabs:** The permission **tabs** [21] enables the extension to manage the *browser tabs and windows* in the client-browser. This includes the creation and modification of tabs and windows. The permission is used by the SSP-Extension to identify the correct tab, which wants to establish the Secure Session. This is needed to not interfere other opened tabs. Further, the tabs permission is used for creating the Secure Tab, when the Secure Set is successfully exchanged.

- **webRequest:** Using the **webRequest** [20] permission, enables custom browser extensions to analyse network traffic and modify the traffic on the fly. The API gives the developer different possibilities to access the traffic at a certain point of time. In figure 25 the life cycle of an incoming and outgoing web request is visualized. For any shown state of the life cycle, the developer has the possibility to set up a listener to access the web packet at the given point of execution.

  The webRequest permission is used for several actions of the Secure Session Protocol. The information exchanged between the SSP-Extension and the web service is injected into the web packets of the client-browser. Further, the Secure Session Protocol uses the webRequest API to append the Secure Session Protocol HTTP header. Security features like the *(extended-) Content Security Policy* and the *Integrity Check* of the web packets during the **Running** state are using the webRequest API.

  Additionally to the webRequest permission, the SSP-Extension uses the **webRequestBlocking** permission, which enables the SSP-Extension to completely block certain incoming and outgoing packets. This permission is needed to enforce the *(extended-) Content Security Policy*.

- **storage**: To store and load Secure Session Protocol related data, the **storage** permission [19] of the Platform API is used. This permission enables the extension to store information within a reserved memory location. Information about the web services is stored to directly establish the Secure Session with an already paired web service. This includes the Secure Session Key, the unique client-ID and the URL of the web service. Nevertheless, as stated in the description of the API, this storage location **should not** be used for confidential data [19]. This problem of the current implementation is addressed in subsection 6.4 of this chapter.

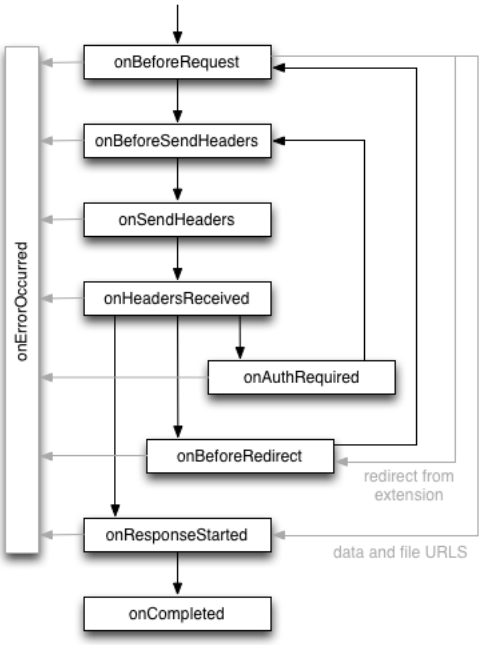

Figure 25: Life-cycle of a web request

- **proxy**: The **proxy** API [18] enables the extension to check the proxy configuration of the browser. This permission is solely used by the SSP-Extension to report a proxy usage to the web service. Therefore, this permission is only used within the *Collecting* phase of the **Establishing** state.
- **management:** The **management** API [17] allows custom browser extension to manage other installed and running custom browser extensions (and apps). This permission is used by the SSP-Extension in two ways. First, to achieve a maximum level of the security, the SSP-Extension needs to be installed correctly and must check that certain permissions of the Platform API are not used by other extensions. This includes the permission management. If an extension, besides the SSP-Extension, obtains the permission management, it might be possible that this custom browser extension deactivates the SSP-Extension during the execution of the Secure Session. Further, extension with the permission webRequest (described above) might smaller the achieved security, due to the fact that other extension also try to modify the web packets. If the SSP-Extension finds extensions with one of the two mentioned permissions, the extension prompts the user to remove those extensions in order to use the Secure Session Protocol correctly.

  Besides the security checks during the installation, the management permission is also used for enforcing a security feature of the Secure Set. If specified by the web service, the SSP-Extension is able to deactivate all installed custom browser extensions for the runtime of the Secure Session. During the execution of the Secure Session, the extension then is able to control the installed custom browser extensions and hinder a manual activation of a custom browser extension during the Secure Session. After the Secure Tab is closed, the SSP-Extension is able to activate the disabled custom browser extensions again.

Background Page and Browser Action

The backgroundPage and the browserAction (both written in JavaScript) are using third-party libraries to execute the Secure Session Protocol. For de- and encrypting the traffic between the web service and the SSP-Extension, the background page uses the cryptographic library *slowAES* [52], which provides standardized cryptographic algorithms. In addition, the library *BigInt.js* by Leemon Baird [4] is used for calculating the Secure Session Key by executing the Diffe-Hellmann Key Agreement. As for the web service, the browserAction includes the Bootstrap CSS3 files for visualization purposes.

6.3 Details of the Implementation

The following section describes all features currently implemented through the SSP-Extension. The features are listed chronologically as they are appearing during the Secure Session Protocol execution.

During the **Ready** state of the Secure Session Protocol, the SSP-Extension uses the *webRequest* API to inspect any outgoing web request by the client-browser. Further, the SSP-Extension adds the Secure Session Protocol HTTP header field. This is done across all client-browser tabs. If the SSP-Extension identifies an already paired web service, it informs the web service by sending an encrypted web packet (including the client-ID of the web service) to the web service. The information about already paired web services is stored within the browser, by using the storage permission.

The **Pairing** step is fully implemented in the current version of the Secure Session Protocol Implementation. The SSP-Extension uses the libraries *slowAES* and *BigInt.js* to calculate and execute all relevant steps required for the Secure Session Key. The triple of Secure Session Key, client-ID and web service URL is stored within the local storage of the SSP-Extension. After the calculation is done, the *slowAES* library is used for de- and encrypting the traffic of the covert channel. Therefore, the methods supplied by Kevin Kutcha are used, as they are supporting symmetric encryption and decryption in JavaScript and PHP [34].

Currently implemented in the **Establishing** state of the protocol is the execution of the *Collecting* phase. The SSP-Extension is able to collect correct information about the client-browser. In the current version of the implementation, the SSP-Extension collects the following client-side information:

- **User Agent:** The browser vendor and version is collected by the SSP-Extension through the JavaScript **navigator** object. The field *navigator.userAgent* returns the current version of the installed browser and the *navigator.vendor* field returns the vendor of the browser.
- **Extensions:** By using the *management* permission of the Platform API, the SSP-Extension is able to extract a list of all installed and running custom browser extensions.
- **Plug-Ins:** The different names and versions of installed third-party plug-ins is collected like the User Agent through the JavaScript *navigator* object. All information can be extracted through the field *navigator.plugins*. For testing purposes, only information about the third-party plug-ins *Silverlight, Flash Player and Java Runtime* are collected.
- **Network Status:** The *proxy* permission of the Platform API is used to collect the current network status. This permission enables the SSP-Extension to identify, if the network connection currently is behind a proxy. ***Note: A proxy is only detectable through this permission, if the proxy is set in the network connection of the operating system. Transparent proxies are currently not supported by the Secure Session Protocol.***

Based on the encrypted information from the SSP-Extension, the web services evaluates the Secure Set for the Secure Session. The next step at the client is the *Building* phase executed by the SSP-Extension.

Through the *tabs* permission of the Platform API, the extension creates the Secure Tab. Currently the Secure Tab is displayed within a new window, which is created as a *popup*. In figure 26 the difference between a normal browser window and a popup window is shown. The popup window is slimmed browser window. The navigation bar at the top is hidden. Only a bar with the non-modifiable URL is displayed. On the left the web service is displayed without the Secure Session Protocol. On the right, the web service is displayed by using the Secure Session Protocol.

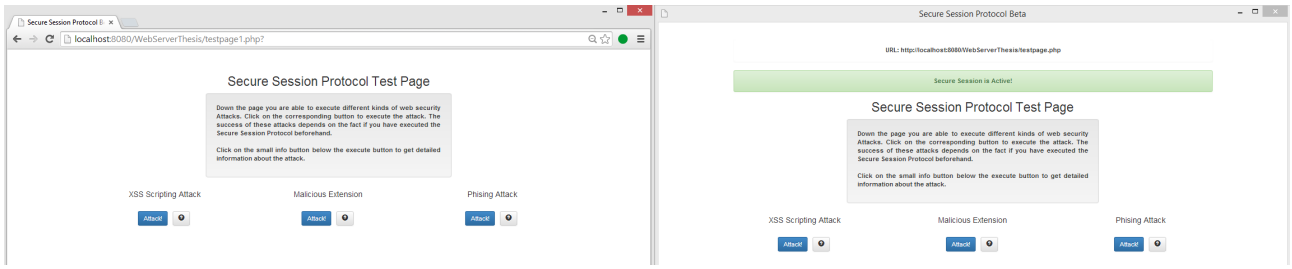

Figure 26: Difference between a normal and popup window (Chrome Platform API)

To enforce the extension parameter, the SSP-Extension uses the *management* API to manage installed and running plug-ins. If the private browsing parameter is set to true, within the Secure Set, the extension can specify the newly created Secure Tab to be created in incognito mode (Google Chrome specific).

After the SSP-Extension has started the Secure Session, by creating the Secure Tab, the SSP-Extension, as mentioned in the concept chapter of this work, switches to the observing mode. During the Running state of the protocol, the

SSP-Extension uses the *management* API to control the status of all installed extensions. No extension is allowed to be activated or installed manually during the execution of the Secure Session. Therefore, the SSP-Extension forbids all actions concerning custom browser extensions. Further, to ensure that the (Extended-) Content Security Policy is enforced, the SSP-Extension monitors all outgoing and incoming network resources by using the *webRequest* API. In addition, the SSP-Extension checks the integrity of the web packets received by the web service. This is done by using the hash function SHA-256, included in the Crypto JS [8] library.

The Secure Tab was created with the *tabs* permission of the Platform API. Therefore, the background JavaScript page is able to detect the termination of the Secure Session as soon as the Secure Tab is closed. After the Secure Session is closed, the SSP-Extension needs to restore all properties of the browser as they were before the establishment of the Secure Session. This includes reactivating the custom browser extensions and resetting the Secure Session Protocol. The SSP-Extension changes back into the Ready state, waiting for the next incoming web service with Secure Session Protocol support.

6.4 Improvements

The current implementation of the Secure Session Protocol is based on a custom browser extension. The main goal of this work is to underline the importance of an improved web security technique. The presented implementation is used for conceptually testing the concept of the Secure Session Protocol. Therefore, as it was mentioned in the introduction of this chapter, the implementation is not intended to be used in a productive environment, because the intended security level cannot be achieved. This section shows missing function and drawbacks of the current concept implementation. The section concludes with an outlook and advises for a productive implementation.

- **Access to the HTTPS certificates:** The SSP-Extension currently has no possibility to access received HTTPS certificates. The Google Chrome JavaScript Platform API offers no possibility to computational access information of HTTPS certificates. Explained in section 4, checking the HTTPS certificate is necessary to avoid a Man-In-The-Middle attack, who tries to spoof the HTTPS connection in both directions (so called *SSL-Stripping Attacks*). To ensure the correctness of the HTTPS connection, the Secure Session Protocol needs access to the delivered HTTPS certificates. This can be achieved by implementing the SSP-Extension within the browser.

- **Secure Storage possibilities:** For storing information about paired web services, the storage permission of the Google Chrome JavaScript Platform API is used. The current implementation uses the storage location to store the symmetric Secure Session Keys, which are highly relevant for the security level of the protocol. Stated on the documentation page of the storage API, the storage location should not be used for confidential data, because the memory location is not encrypted [19]. Further, the storage location is shared by all custom browser extension and therefore can easily be accessed by other extensions.

  One possible solution is to store no information about the paired web services. Therefore, no confidential data is stored on the client-side. A major drawback of this approach is that the **Pairing** state is executed before each Secure Session. As seen in chapter 3.3.2, this property can be intended by some web services, but the majority of the web services wants the **Pairing** state to be executed only once. Shown at the end of this subsection, this problem can be solved by implementing the SSP-Extension within the browser.

- **Controlling installed extensions:** A further problem of the current implementation arises from the design decision of the implementing the SSP-Extension as a custom browser extension. Part of the security concept is to manage installed and running custom browser extension. The *management* permission allows custom browser extensions to enforce this policy. To avoid a deactivation of the SSP-Extension, no other custom browser extension is permitted to use the management permission, because a potential adversary could easily deactivate the SSP-Extension.

  Permitting only extension without the *management* API is a drawback in regards towards usability, because the user is restricted to only install custom browser extensions without the *management* permission. Further, the current solution does not provide full security. An adversary could nevertheless disable the SSP-Extension manually.

  To solve this problem, the mechanism to manage custom browser extension needs to be protected from other custom browser extensions. This can be achieved by implementing the SSP-Extension within the browser.

- *Controlling installed plug-ins:* The current implementation of the protocol gathers information about installed third-party plug-ins. This information is used for checking the proper version of the plug-in, e.g., to inform the user about an outdated plug-in. The complete protocol, as described in chapter 3, also enables the web service to

specify a list of third-party plug-ins which should be disabled throughout the Secure Session. This mechanism is not supported by custom browser extension and thus, the current implementation is unable to support this feature.

Deactivating third-party plug-ins is one of the key security features of the Secure Session Protocol. By implementing the SSP-Extension within the browser, this issue can be solved.

Implementation within the browser

The current implementation of the Secure Session Protocol misses several features which are crucial for the achieved security level of the protocol. The current limitation of the protocol arises from the fact, that the implementation is based on a custom browser extension. Therefore, relevant functions to fully execute the Secure Session Protocol are not available. As mentioned in the previous listing, all drawbacks can be solved by implementing the protocol within the browser.

Implementing the SSP-Extension as a part of the web browser, enhances the security level. The missing features of the current implementation (HTTPS certificate access, controlling extensions, controlling plug-ins and secure storage possibilities) can all be implemented, if the protocol is executed as a part of the browser. Further, currently implemented concepts can be enhanced and new security features are possible to be implemented.

The installation process of the SSP-Extension is obsolete. The SSP-Extension is implemented in the base installation of the browser and therefore the extension does not need to check for potential malicious custom browser extension, which are able to mitigate the execution of the Secure Session Protocol. Further, the usability of the client-browser is no longer harmed, because all customer browser extensions can be installed without restrictions.

In addition, the security concept of the Secure Session Protocol can be expanded, to not only check browser specific information, but also collect information related to the underlying operating system. This feature includes checks of the current virus signature or the installation of certain tools.

Seen in this section, the current implementation of the Secure Session Protocol is not usable for productive environments. This is due to the fact, that the current implementation of the SSP-Extension is implemented as a custom browser extension. Therefore, the SSP-Extension is not able to execute the Secure Session Protocol as specified in the concept and thus the security model cannot be realized correctly. To achieve the desired security level, the Secure Session Protocol needs to be implemented as a part of the client-browser.

7 Future Work

In this section, additional and further enhancements of the protocol are presented. First, the sections starts with a concept, which enables the web service to decide which information of the client-browser the SSP-Extension needs to collect. Second, the feature of enforcing the Secure Set is used in another use-case, unrelated to security.

Improved Collecting phase: To further strengthen the security on the client-side and to decrease the overhead, which is produced by the Secure Session Protocol, an improved *Collecting* phase can be implemented. Currently, the SSP-Extension on the client-side collects pre-defined information about the client-browser (see section 3.3.3 for more information). For certain web services, some of the information are useless, because they are not relevant for the security checks and for the Secure Set. Therefore, additional overhead is produced as unused information are sent to the web service. This overhead can be mitigated, if the web service is able to specify the information needed for the *Creating* phase. All needed information can be exchanged during the *Pairing* phase and can be locally stored together with the pairing parameters.

As on the one hand this mechanism reduces the produced overhead of the Secure Session Protocol, a possible privacy problem could arise from this mechanism, since the web service is ably to arbitrary collect information about the client-browser (and computer, if the protocol is further extended to access the operating system). Therefore, the collecting mechanism should be extended to a *challenge-response protocol*. Instead of sending correct information about the client to the web service, the web service can send a challenge to the SSP-Extension, which is then evaluated on the client-side. For example, currently the web service checks the version of the client-browser by receiving the complete user-agent string. Then, the web service extracts the version from the string and matches the version to the policy requirements. In an improved version of the *Collecting* phase, the web service would send a challenge scheme to the SSP-Extension, asking the SSP-Extension to evaluate the query. An example of the challenge-response protocol improvement is shown in figure 28. As the SSP-Extension is a trustworthy partner of the web service, the SSP-Extension will answer correctly. The mentioned technique was introduced in [6].

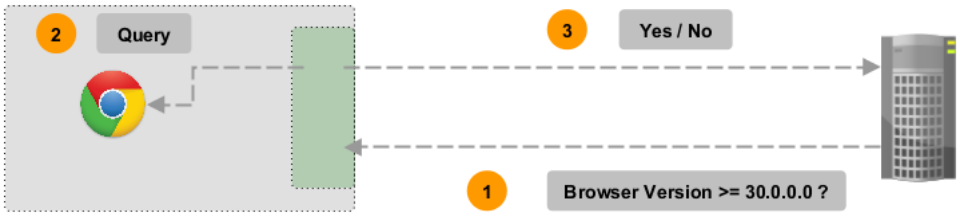

Figure 27: Example of the Challenge-Response Improvement

The SSP-Extension would answer with the values Yes or No, indicating that the query is passed or failed. This mechanism increases the privacy of the client-user as no information about the client-browser (and computer) are transmitted to the web service.

As a second outlook for the usage of the Secure Session Protocol, a different use-case, not related to security, is described. In this example, the web service uses the **enforcement feature** of the Secure Session Protocol to ensure, that the web page is delivered to the user as intended. In this example, we consider the web service to be a video platform, e.g., YouTube [27]. The main source of income of (most) such web services is advertisement [35]. The advertisement is used on the web page, at the beginning of videos and within the videos. Because of that, users are disturbed by the placement of the advertisement and are therefore using tools that are blocking advertisement on selected web services. These tools are called *AdBlocker*. They are delivered as custom browser extension and are freely available for the user. After installing an *AdBlocker*, the user is able to specify web services on which no advertisement should be displayed and therefore the advertisement gets blocked.

As on the one hand, *AdBlocker* tools are increasing the user experience, they on the other hand decrease the income of web services. Therefore, such web services could use the Secure Session Protocol to deliver the web service (with advertisement) as it was intended by the web service provider. The web service could determine in the Secure Set, that throughout the Secure Session, all custom browser extensions (including the AdBlocker extension) should be disabled.

8 Conclusion

In the introductory section of this work, the current problem of security related web services is described. Despite the introduction of new web security techniques, as described in chapter 2, the number of successfully executed web security attacks increases. This can be explained by fact that web services are not able to verify that the created web security rules are enforced on the client-side. From the perspective of the web service, the client-browser is a **blackbox** as described in section 1.3.

The Secure Session Protocol introduces an advanced web security concept. It enables the web services to **enforce security policies** on the client-side. This property is achieved by installing a **trustworthy partner** of the web service at the client-side. The so called SSP-Extension further enables the web service to collect correct information about the client-browser. The communication partners are exchanging messages through an established covert channel, which is secured with the Secure Session Key.

Comparing figure 6 from section 1.3, the Secure Session Protocol is together with the SSP-Extension *able to shed light into the blackbox*. The client-browser is no longer a blackbox for the web service. This is due to the fact, that the SSP-Extension is a trustworthy partner of the web service on the client-side.

Figure 28: Result of the Secure Session Protocol.

In chapter 7, the possibility for web services to misuse the Secure Session Protocol for not security related use cases is discussed. The features of the Secure Session Protocol can be potentially misused by other not mentioned use cases or possibly by adversaries. Therefore, an extension to the current version of the Secure Session Protocol needs to be developed, to mitigate a potential misuse.

Despite the result of the Secure Session Protocol, this work is only a concept and a productive implementation is not ready to be deployed. As described in section 6.4, to use the Secure Session Protocol for security related web services, an implementation within the client-browser is needed. Implementing the SSP-Extension as a part of the client-browser further increases the achieved security of the Secure Session Protocol as the SSP-Extension has access to the underlying operating system.

In order to use the Secure Session Protocol, web service providers have to adapt to their web service to use the Secure Session Protocol. Adaptation of web security mechanism by web services is another mentionable security issue. For example, the newly introduced Content Security Policy, as mentioned in section 4, mitigates many web security risks like Cross-Site-Scripting, Clickjacking and Packet Sniffing. Nevertheless, the Content Security Policy is rarely used by only some web services.

Furthermore, the achieved security of the Secure Session depends, as mentioned in section 3.3.3 and chapter 4, depends on the created Secure Set. If the included security rules are specified correctly, the security on the client-side (and therefore for the whole web service) can be maximized. However, if the web service does not specify the Secure Set properly, the security can be decreased. Therefore, the achieved security of the Secure Session Protocol depends on the web service.

References

[1] Adam Barth Google, Inc., Dan Veditz Mozilla Corporation, Mike West Google, Inc. Content Security Policy 1.1. http://www.w3.org/TR/2014/WD-CSP11-20140211/, 2014. Online-Source; last checked 02-August-2014.

[2] Adobe. Adobe Flash Player. http://www.adobe.com/de/products/flashplayer.html, 2014. Online-Source; last checked 23-July-2014.

[3] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium*, 2010.

[4] BigInt.js 5.5. Big Integer Library v. 5.5. http://www.leemon.com/crypto/BigInt.js, 2013. Online-Source; last checked 03-August-2014.

[5] Bundeskriminalamt (BKA). Bundeskriminalamt. http://www.bka.de/, 2014. Online-Source; last checked 02-August-2014.

[6] J. Camenisch, a. shelat, D. Sommer, S. Fischer-Hubner, M. Hansen, H. Krasemann, G. Lacoste, R. Leenes, and J. Tseng. Privacy and Identity Management for Everyone. In *Proceedings of the 2005 Workshop on Digital Identity Management*, DIM '05, 2005.

[7] Check Point Software Technologies Ltd. Endpoint Security. http://www.checkpoint.com/products/index.html#endpoint, 2014. Online-Source; last checked 02-August-2014.

[8] Crypto-JS. JavaScript implementations of standard and secure cryptographic algorithms. https://code.google.com/p/crypto-js/, 2014. Online-Source; last checked 03-August-2014.

[9] K. Curran and T. Dougan. Man in the browser attacks. *Int. J. Ambient Comput. Intell.*, 2012.

[10] Dyrk Scherff. Die Bank flüchtet ins Netz. *Frankfurter Allgemeine Zeitung - FAZ*, 2013. Online-Source; last checked 02-August-2014.

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, 2012.

[12] Franz Nestler. Mehr Hackerangriffe auf Online-Banking. *Frankfurter Allgemeine Zeitung - FAZ*, 2014. Online-Source; last checked 26-July-2014.

[13] S. Frei, T. Duebendorfer, G. Ollmann, and M. May. Understanding the Web browser threat. Technical Report 288, TIK, ETH Zurich, June 2008. Presented at DefCon 16, Aug 2008, Las Vegas, USA. http://www.techzoom.net/insecurity-iceberg.

[14] Google Inc. App Manifest. http://developer.android.com/guide/topics/manifest/manifest-intro.html, 2014. Online-Source; last checked 02-August-2014.

[15] Google Inc. Background Pages. https://developer.chrome.com/extensions/background_pages, 2014. Online-Source; last checked 26-July-2014.

[16] Google Inc. chrome.browserAction. https://developer.chrome.com/extensions/browserAction, 2014. Online-Source; last checked 01-August-2014.

[17] Google Inc. chrome.management. https://developer.chrome.com/extensions/management, 2014. Online-Source; last checked 01-August-2014.

[18] Google Inc. chrome.proxy. https://developer.chrome.com/extensions/proxy, 2014. Online-Source; last checked 01-August-2014.

[19] Google Inc. chrome.storage API. https://developer.chrome.com/extensions/storage, 2014. Online-Source; last checked 03-August-2014.

[20] Google Inc. chrome.tabs. https://developer.chrome.com/extensions/webRequest, 2014. Online-Source; last checked 01-August-2014.

[21] Google Inc. chrome.tabs. https://developer.chrome.com/extensions/tabs, 2014. Online-Source; last checked 01-August-2014.

[22] Google Inc. Google Chrome 35.0.1916.153m. https://www.google.com/chrome/browser/, 2014. Online-Source; last checked 03-August-2014.

[23] Google Inc. Google Inc. https://www.google.de/intl/de/about/company/, 2014. Online-Source; last checked 03-August-2014.

[24] Google Inc. JavaScript APIs. https://developer.chrome.com/extensions/api_index, 2014. Online-Source; last checked 03-August-2014.

[25] Google Inc. The manifest file. https://developer.chrome.com/extensions/overview#manifest, 2014. Online-Source; last checked 02-August-2014.

[26] Hamilton Ulmer. Understanding Private Browsing. http://blog.mozilla.org/metrics/2010/08/23/understanding-private-browsing/, 2010. Online-Source; last checked 02-August-2014.

[27] G. Inc. YouTube. http://www.youtube.com/, 2014. Online-Source; last checked 03-August-2014.

[28] Internet Engineering Task Force (IETF). Deprecating the X-Prefix and Similar Constructs in Application Protocols. http://tools.ietf.org/html/rfc6648, 2012. Online-Source; last checked 02-August-2014.

[29] IT Law Wiki. Sensitive data. http://itlaw.wikia.com/wiki/Sensitive_data, 2014. Online-Source; last checked 01-August-2014.

[30] ITWissen.info. chipTAN-Verfahren. http://www.itwissen.info/definition/lexikon/chipTAN-Verfahren-chipTAN-Verfahrens-chipTAN.html, 2013. Online-Source; last checked 03-August-2014.

[31] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, 2007. Online-Source; last checked 26-July-2014.

[32] John J. G. Savard. A Cryptographic Compendium. http://www.quadibloc.com/crypto/pk0503.htm, 1998. Online-Source; last checked 02-August-2014.

[33] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor Carnegie Mellon University. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[34] Kevin Kutcha. Matching PHP and JS Encryption. http://kevinkuchta.com/_site/2011/08/matching-php-and-js-encryption/, 2014. Online-Source; last checked 04-August-2014.

[35] M. Lorenz. Wie Google mit Werbung und auf YouTube absahnt. *Wirtschaft Woche*, January 2013. Online-Source; last checked 03-August-2014.

[36] M. Abramowitz, I. A. Stegun. Handbook of Mathematical Functions. http://people.math.sfu.ca/~cbm/aands/abramowitz_and_stegun.pdf, 1972. Online-Source; last checked 21-July-2014.

[37] Martin Bartosch. Good numbers, bad numbers. *The H Security*, 2008. Online-Source; last checked 21-July-2014.

[38] Mike Ter Louw, Jin Soon Lim, V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. http://link.springer.com/article/10.1007%2Fs11416-007-0078-5, January 2008.

[39] Moxie Marlinspike. Stripping SSL To Defeat HTTPS In Practice, Blackhat Europe 2009. https://blackhat.com/presentations/bh-europe-09/Marlinspike/blackhat-europe-2009-marlinspike-sslstrip-slides.pdf, 2009. Online-Source; last checked 29-July-2014.

[40] Mozilla Developer Network and individual contributors. CSP (Content Security Policy). https://developer.mozilla.org/en-US/docs/Web/Security/CSP, 2014. Online-Source; last checked 02-August-2014.

[41] Mozilla Foundation. Private Browsing - Browse the web without saving information about the sites you visit. https://support.mozilla.org/en-US/kb/private-browsing-browse-web-without-saving-info, 2014. Online-Source; last checked 20-July-2014.

[42] Network Working Group. Diffie-Hellman Key Agreement Method. http://www.ietf.org/rfc/rfc2631.txt, 1999. Online-Source; last checked 23-July-2014.

[43] Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt, 1999. Online-Source; last checked 29-July-2014.

[44] Network Working Group. HTTP Over TLS. http://tools.ietf.org/html/rfc2818, 2000. Online-Source; last checked 29-July-2014.

[45] Open Web Application Security Project. Open Web Application Security Project Top 10 2013. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf, 2013. Online-Source; last checked 03-August-2014.

[46] Open Web Application Security Project. Open Web Application Security Project. https://www.owasp.org/index.php/Main_Page, 2014. Online-Source; last checked 03-August-2014.

[47] Oracle Technology Network. Java SE at a Glance. http://www.oracle.com/technetwork/java/index.html, 2014. Online-Source; last checked 01-August-2014.

[48] PHP Group. getallheaders. http://php.net/manual/de/function.getallheaders.php, 2014. Online-Source; last checked 03-August-2014.

[49] PHP Group. PHP 5.5.11 Release. http://php.net/releases/5_5_11.php, 2014. Online-Source; last checked 29-July-2014.

[50] PHP Secure Communications Library. PHP Secure Communications Library. http://phpseclib.sourceforge.net/math/intro.html, 2014. Online-Source; last checked 29-July-2014.

[51] PHP Secure Communications Library. PHP Secure Communications Library. http://phpseclib.sourceforge.net/, 2014. Online-Source; last checked 29-July-2014.

[52] slowAES. slowAES for PHP and JavaScript. https://code.google.com/p/slowaes/, 2014. Online-Source; last checked 01-August-2014.

[53] Sonatype. Executive Brief: Addressing Security Concerns in Open Source Components. http://img.en25.com/Web/SonatypeInc/%7Bb2fa5ed8-938d-4bce-8a9c-d08ebeba826d%7D_Executive_Brief_-_Study-_Understanding_Security_Risks_in_OSS_Components-1.pdf, 2013. Online-Source; last checked 29-July-2014.

[54] Sparkasse. *Online-Banking mit smsTAN*, 2014. Online-Source; last checked 03-August-2014.

[55] Statista GmbH. 1 Milliarde verkaufte Smartphones in 2014. http://de.statista.com/infografik/1012/prognose-absatze-von-connected-devices/, 2014. Online-Source; last checked 26-July-2014.

[56] Statista GmbH. Anteil der Internetnutzer, die auch mobiles Internet verwenden in den einzelnen Altersgruppen in Deutschland. http://de.statista.com/statistik/daten/studie/197417/umfrage/mobile-internetnutzung-in-deutschland-in-den-altersgruppen/, 2014. Online-Source; last checked 29-July-2014.

[57] Statista GmbH. Anteil der Nutzer von Online-Banking in Deutschland in den Jahren 1998 bis 2013. http://de.statista.com/statistik/daten/studie/3942/umfrage/anteil-der-nutzer-von-online-banking-in-deutschland-seit-1998/, 2014. Online-Source; last checked 02-August-2014.

[58] Statista GmbH. Anzahl der Internetnutzer weltweit von 1997 bis 2013 (in Millionen). http://de.statista.com/statistik/daten/studie/186370/umfrage/anzahl-der-internetnutzer-weltweit-zeitreihe/, 2014. Online-Source; last checked 03-August-2014.

[59] Statista GmbH. Ist Ihnen schon einmal ein finanzieller Schaden im Zusammenhang mit Online-Banking entstanden? http://de.statista.com/statistik/daten/studie/4383/umfrage/finanzieller-schaden-im-zusammenhang-mit-online-banking/, 2014. Online-Source; last checked 12-July-2014.

[60] Terri Oda, Anil Somayaji. Enhancing Web Page Security with Security Style Sheets. http://terri.zone12.com/doc/academic/TR-11-04-Oda.pdf, 2011. Online-Source; last checked 26-July-2014.

[61] The Apache Software Foundation. HTTP Server Project. https://httpd.apache.org/, 2014. Online-Source; last checked 03-August-2014.

[62] The Mozilla Foundation. How Many Firefox Users Customize Their Browser? http://blog.mozilla.org/metrics/2009/08/11/how-many-firefox-users-customize-their-browser/, August 2009. Online-Source; last checked 03-August-2014.

[63] The Mozilla Foundation. The Mozilla Foundation. https://www.mozilla.org/en-US/foundation/, 2014. Online-Source; last checked 03-August-2014.

[64] The Open Web Application Security Project. Man-in-the-middle attack. https://www.owasp.org/index.php/Man-in-the-middle_attack, 2009. Online-Source; last checked 29-July-2014.

[65] Trend Labs. *When Android Apps want more than they need*, 2011. Online-Source; last checked 03-August-2014.

[66] Twitter Bootstrap 3.2.0. Bootstrap JS, HTML5 and CSS3 Framework. http://getbootstrap.com/, 2014. Online-Source; last checked 03-August-2014.

[67] W3C. Cascading Style Sheets. http://www.w3.org/Style/CSS/, 2014. Online-Source; last checked 23-July-2014.

[68] Wikipedia, the free encyclopedia. Out-of-band. http://en.wikipedia.org/wiki/Out-of-band, 2014. Online-Source; last checked 29-July-2014.