Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Mariana Alexandra Aleixo de Barcelos

Desenvolvimentos para uma plataforma de segurança no ambiente da cadeia de fornecimento de semicondutores

Towards a security framework for the semiconductor supply chain environment

Mariana Alexandra Aleixo de Barcelos

Desenvolvimentos para uma plataforma de segurança no ambiente da cadeia de fornecimento de semicondutores

Towards a security framework for the semiconductor supply chain environment

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações

Apoio financeiro da
Fundação para a Ciência e a Tecnologia

o júri / the jury

presidente / president

Professor Doutor Armando Humberto Moreira Nolasco Pinto
Professor Associado da Universidade de Aveiro

Professor Doutor Rogério Pais Dionísio
Professor Adjunto da Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco

Doutor Jonathan Rodriguez Gonzalez
Professor Associado Convidado da Universidade de Aveiro (orientador)

acknowledgements

The work has been performed in the project Power Semiconductor and Electronics Manufacturing 4.0 (SemI40), under grant agreement No 692466. The project is co-funded by the Fundação para a Ciência e Tecnologia (ECSEL/0009/2015) and Electronic Component Systems for European Leadership Joint Undertaking (ECSEL JU).

Palavras-chave Secure communications, SSL/TLS, OAuth 2.0, OpenID Connect, Semiconductor Supply Chain

Resumo

Hoje em dia, a troca de informação entre os parceiros da cadeia de forne-cimento de semicondutores pode ser alvo de muitas ameaças de segurança conhecidas e desconhecidas no ambiente interno/externo dos parceiros. Particularmente, estas vulnerabilidades, no ambiente da cadeia de fornecimento de semicondutores, podem ser exploradas por atacantes com um amplo espectro de motivações que vão desde intenções criminais, visando o ganho financeiro, até à espionagem industrial e a cyber-sabotagem. Os atacantes podem comprometer a comunicação de dados entre parceiros na cadeia de fornecimento e, portanto, podem prejudicar o fornecimento de serviços pelos parceiros, bem como a continuidade da prestação de serviços. Como resultado, os parceiros da cadeia de fornecimento de semicondutores poderão sofrer repercussões nocivas que podem causar perdas significativas de receita, destruição da sua marca e atrasos no avanço das suas tecnologias. Consequentemente, uma plataforma de segurança para o ambiente da cadeia de fornecimento de semicondutores é de extrema importância. Assim, a intenção desta tese é fornecer uma base para uma plataforma de segurança para comunicação segura de dados entre todos os parceiros da cadeia de fornecimento de semicondutores.

Keywords Secure communications, SSL/TLS, OAuth 2.0, OpenID Connect, Semiconductor Supply Chain

Abstract Nowadays, data communication across the partners in the semiconductor supply chain can be the target of many known and unknown security threats exploiting security vulnerabilities in the internal/external environment of the partners. Particularly, these vulnerabilities in the semiconductor supply chain environment can be exploited by attackers with a wide spectrum of motivations ranging from criminal intents aimed at financial gain to industrial espionage and cyber-sabotage. Attackers can compromise the data communication between legitimate parties in the supply chain and thus can jeopardize the delivery of services across the partners as well as the continuity of the service provision. As a result, semiconductor supply chain partners will suffer from damaging repercussions which can cause significant revenue loss, destroy their brand and eventually hinder their advancement. Consequently, a security framework for the semiconductor supply chain environment is of utmost importance. Hence, the intent of this thesis is to provide a foundation for a security framework for secure data communication across the partners in the semiconductor supply chain.

Contents

List of Figures

Acronyms

BEAST Browser Exploit Against SSL/TLS

BREACH Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext

CBC Cipher Block Chaining

CRIME Compression Ratio Info-leak Made Easy

CSRF Cross-Site Request Forgery

DoS Denial of Service

FTP File Transfer Protocol

HSTS HTTP Strict Transport Security

IAM Identity and Access Management

ICs Integrated Circuits

ICS industrial control systems

LDAP Lightweight Directory Access Protocol

MAC Message Authentication Code

MITM Man-In-The-Middle

NNTP Network News Transfer Protocol

POODLE Padding Oracle On Downgraded Legacy Encryption

POP3 Post Office Protocol 3

PRGA Pseudo-random Generation Algorithm

RAT Remote Access Tools

RSA Rivest-Shamir-Adleman

SMTP Simple Mail Transfer Protocol

SSL Secure Sockets Layer

TIME Timing Info-leak Made Easy

TLS Transport Layer Security

XMPP Extensible Messaging and Presence Protocol

Chapter 1

Introduction

1.1 Motivation

Nowadays, data communication across the partners in the semiconductor supply chain can be the target of many known and unknown security threats exploiting many security breaches in the internal/external environment of the partners due to its heterogeneous and dynamic nature as well as the fact that non-professional users in security issues usually operate their information systems. Particularly, these vulnerabilities in the semiconductor supply chain environment can be exploited by attackers with a wide spectrum of motivations ranging from criminal intents aimed at financial gain to industrial espionage and cyber-sabotage. Attackers can compromise the data communication between legitimate parties in the semiconductor supply chain and thus can jeopardize the delivery of services across the partners as well as the continuity of the service provision. As a result, semiconductor supply chain partners will suffer from damaging repercussions, which can cause significant revenue loss, destroy their brand and eventually hinder their advancement. Consequently, a security framework for secure data communications across the partners in the semiconductor supply chain environment is of utmost importance.

1.2 Objectives

This thesis aims to provide a foundation for a security framework for secure data communications across the partners in the semiconductor supply chain. Towards this direction, we firstly provide an overview of the semiconductor supply chain environment along with the description of the functionality of its main components. Furthermore, we consider representative examples of various attacks that have been seen in the wild and can cause potential security issues and challenges in the semiconductor supply chain environment. The range of the attacks shows how vital is a security framework for secure data communications for the partners in the supply chain of the semiconductor industry. Moreover, in the context of this thesis, we aim to study the SSL/TLS protocol which is the de facto standard for secure Internet communications [25], [37], [12] and the OAuth 2.0 protocol

which is the industry-standard protocol for authorization [42], [15]. Both the SSL/TLS protocol and the OAuth 2.0 protocol are two standard security protocols that can be essential components of a security framework, providing mechanisms to ensure security in communication between the partners in the semiconductor supply chain that can be the target of many known and unknown attacks. However, the SSL/TLS protocol and the OAuth 2.0 protocol are vulnerable against a number of attacks. Thus, we intend to conduct a review of contemporary literature in order to understand better these two standard security protocols and give an overview of the most well-known attacks against them. The in-depth understanding of these two security protocols will allow us, as a future work towards a security framework for secure data communications, to use them as the basis for the design and implementation of more sophisticated security mechanisms that can address the specific security challenges of the semiconductor supply chain in a more efficient and effective manner. Finally, we target to study the key concepts of the Identity and Access Management discipline that can be an essential component for the security framework since it can enable the right individuals to access the right resources at the right time and for the right reason.

1.3 Contribution

In the context of this thesis, our contribution is the following:

- We gave an overview of the semiconductor supply chain environment along with the description of the functionality of its main components.
- We provided representative examples of various attacks that have been seen in the wild and can cause potential security issues and challenges in the semiconductor supply chain environment.
- We provided detailed description of the SSL/TLS, OAuth 2.0 and OpenID Connect (i.e., an authentication layer on top of the OAuth 2.0).
- We provided the key concepts of the Identity and Access Management discipline.
- We built a virtual security lab, using Virtual Machines (i.e., VirtualBox), where we implemented two scenarios over TLS. We captured the exchanged messages with the network sniffing software Wireshark and then examined them in order to get a better understanding of how SSL/TLS works.
- We implemented a third scenario on a host machine, where we used the Keycloak software, an open source identity and access management solution, in order to get a better understanding the key concepts of the identity and access management discipline and how OpenID Connect works.

1.4 Thesis Outline

Chapter 2 provides the reader with an overview of the semiconductor supply chain environment accompanied by the description of the functionality of its main components. Furthermore, Chapter 2 gives representative examples of various attacks that have been witnessed in the wild and can cause potential security issues and challenges in the semiconductor supply chain environment. In Chapter 3, the description of the SSL/TLS, OAuth 2.0 and OpenID Connect is provided. Moreover, an overview of the most well-known attacks against SSL/TLS and OAuth 2.0 is also given. In Chapter 4, the key concepts of the Identity and Access Management discipline are discussed. Chapter 5 contains the implementation of the scenarios studied in the scope of this thesis. Finally, Chapter 6 summarizes the work done in this thesis.

Chapter 2

Cybersecurity Issues and Challenges in Semiconductor Supply Chain Environment

This Chapter, firstly, provides an overview of the semiconductor supply chain environment along with the description of the functionality of its main components. Furthermore, we consider representative examples of various attacks that have been seen in the wild and can cause potential security issues and challenges in the semiconductor supply chain environment. Moreover, we provide a categorization of the various attack examples based on the intrusion method that they use to compromise the target and gain a persistent foothold in the target's environment.

2.1 Semiconductor Supply Chain

Nowadays, the mass production of Integrated Circuits (ICs), from wafers of raw silicon to the completed chips contained in most electronic devices, is one of the most complex production processes in existence. In addition, semiconductor manufactures do not only need to produce a quality product but also to produce it within a specific time-frame required by the customer [74], [66]. The mass production of ICs can be partitioned into four distinct phases: fabrication, probing, assembly, and final test. Fabrication is the process of transforming a virgin silicon or gallium arsenide wafer, provided by the semiconductor materials suppliers, into a wafer with completed ICs. Moreover, fabrication is the most complex portion of the entire process, since it often requires 300 to 700 different steps. Besides, other factors contributing to fabrication's complexity are the following: re-entrant flows, leading-edge-of-technology yet unreliable equipment, sequence- and part-specific setup times, and competition for capacity with engineering lots. After fabrication, the next step is probing, where the individual ICs on each wafer are tested for basic functionality. After probing, wafers go to the inventory, where they wait for the next two phases: assembly and final test [66]. In the assembly phase, individual ICs are extracted from the wafer and the failed ones

are discarded. Afterwards, the functional ICs are packaged wherein connections are made between the chip and the lead frame. There are several types of packaging, such as plastic or ceramic, through-hole, surface mount, chip carrier, pin-grid arrays, among others. Then, the whole circuit is encapsulated for protection and the packaged ICs move to the final test phase, where they are tested, rated, and ultimately date-stamped for inclusion in the inventory of distributor. The main objective of the final test phase is to ensure that the customer will receive a defect-free product operating properly [98], [66]. It is very common to name the fab phase and probing phase as the "front-end" process, while the assembly phase and final test phase are named as the "back-end" process [98], [61], [66]. In principle, the "front-end" process includes expensive equipment and stringent environmental conditions. Due to the size of ICs, most operations in the "front-end" process have to be done in a clean room as a single speck of dust can damage a microcircuit. In addition, the temperature and humidity also have to be controlled. Moreover, the "front-end" process is performed in-house and takes from 8 to 10 weeks for completion. On the other hand, the "back-end" process is generally contracted to an outside vendor and takes from 1 to 4 weeks for completion [66], [9]. Finally, figure 2.1 shows the information flow among the 7 main components of the Semiconductor Supply Chain environment: a) suppliers, b) front-end, c) inventory, d) back-end, e) distributor, f) customers, and g) customers' customers.

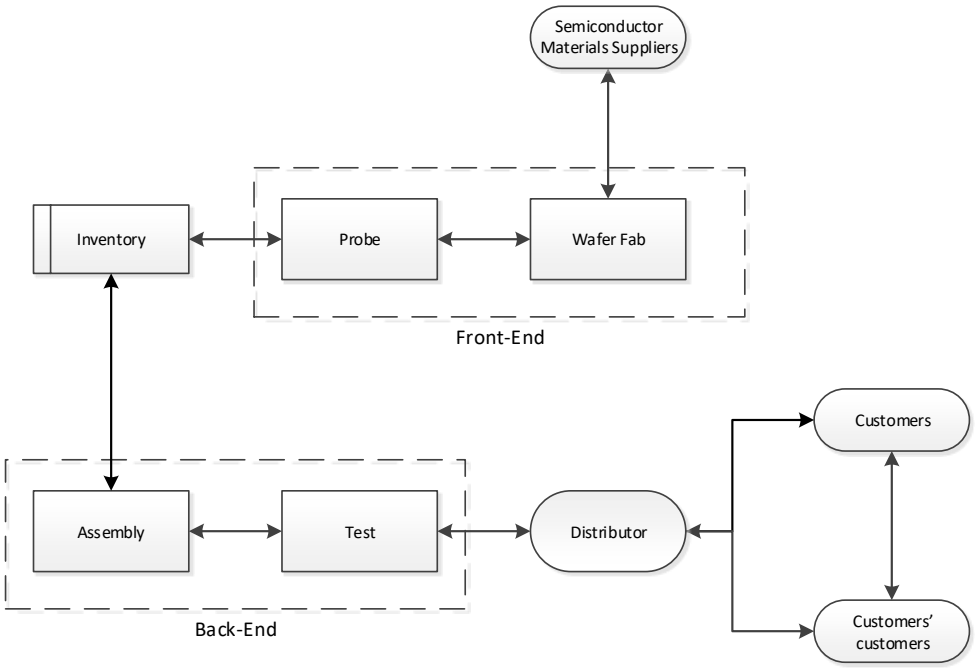

Figure 2.1: Semiconductor supply chain information flow.

2.2 Cyberattacks in Supply Chain

In this section, we consider representative examples of various attacks that have been seen in the wild and can cause potential security issues and challenges in the semiconductor supply chain environment. We categorize these attack examples into 5 main categories based on the intrusion method that they use to compromise the target and gain a persistent foothold in the target's environment. The 5 main categories that we identified are the following: a) spear phishing attacks, b) watering hole attacks, c) attacks based on "trojanized" third-party software, d) attacks based on malicious code and counterfeit certificates, and e) attacks based on tampered devices.

2.2.1 Spear Phishing

Phishing is a kind of social-engineering attack where adversaries use spoofed emails to trick people into sharing sensitive information or installing malware on their computers. Indeed, victims perceive these spoofed emails as being associated with a trusted brand. In other words, phishing attacks target the people using the systems instead of targeting directly the systems that people use. Thus, phishing attacks are able to circumvent the majority of an organization's or individual's security measures. Moreover, it is worthwhile to mention that phishing has spread beyond email to include VOIP, SMS, instant messaging, social networking sites, and even massively multiplayer games. Moreover, cyber-criminals have shifted from sending mass-emails, hoping to trick anyone, to more sophisticated but also more selective "spear-phishing" attacks that use relevant contextual information to trick specific groups of people. In principle, "spear-phishing" attacks are more dangerous than typical phishing attacks [45], [14]. Here are a few examples of "spear-phishing" attacks from the wild.

Icefog

In 2011, Kaspersky Lab started to investigate a threat actor called 'Icefog' that attacked many different groups, such as government institutions, military contractors, telecom operators, satellite operators, among others, through their supply chain. This campaign targeted organizations mostly in South Korea and Japan, but it was suspected that it also targeted the United States and Europe [39], [62]. The intrusion method of this attack was phishing e-mails with a malicious attachment or a link to an infected web page. The attacker could compromise the victim's machine either by tricking the victim to install the attached malware or by tricking the victim to visit the malicious web page [68]. Afterwards, the attacker could steal files from the victim's machine, run commands to locate and steal specific information from the victim's machine, and also communicate with local database servers in order to steal information from them. In addition, Icefog was capable of uploading special tools to extend the capabilities of the installed malware, such as tools for stealing cached browser passwords in the infected machine. In 2012, a Mac OS version of Icefog (Macfog) was created [40], but Kaspersky suspected that it was a beta-testing

phase to be used in targeted victims later. Finally, it is worth mentioning the hit and run nature of Icefog, since the Icefog attackers appeared to know very well what they need from the victims and thus, once the information was obtained, the victim was abandoned [62].

Target

At the end of 2013, Target suffered a cyber-attack that exposed approximately 40 million debit and credit card accounts [56] and 70 million e-mail addresses, phone numbers and other personal information [18]. The hackers started their attack by sending phishing e-mails, including malware, to employees of a third-party vendor, but it was not known if only one vendor was targeted. In addition, it was suspected that the malware in question was Citadel, a password-stealing bot that was a derivative of the ZeuS banking trojan and allowed the attackers to access Target's network by using stolen credentials. It was estimated that the phishing campaign had started at least two months before the main attack carried out [57]. Brian Krebs was the first to break the news about this attack on his security blog [55] followed by Target's Statement, released a day after [97].

Home Depot

In April 2014, just four months after the Target attack, Home Depot was the victim of a data breach. However, they only started investigations on September 2nd, 2014 and released a statement on September 8th, 2014 [43]. It was found that the attackers, similar to the attackers of Target attack, used third party vendor's credentials to access Home Depot's network. After being inside the retailer's network, the attackers exploited a known vulnerability in Windows XP called "zero-days" in order to escape detection [43], [90]. Finally, this attack resulted in the theft of 53 million e-mail addresses and 56 million credit card accounts.

German Steel Mill

In late 2014 (no specific date was provided), Germany's Federal Office for Information Security (BSI) released a report communicating that a German steel mill had been attacked. The attackers' point of entry was the plant's business network and the infiltration was made possible with a spear phishing attack [60]. The phishing emails could have had a malicious attachment or a link to a website from where malware could be downloaded. Once the malware was installed, the attackers were able to take control of the production software. SANS Institute provided the BSI's report, translated to English, where it is mentioned that the attack resulted in an incident where the furnace could not be shut down properly, and as a result, it led to a "massive damage" to the German steel mill [60], [51].

Dragonfly - 1st tactic

A cyber-espionage group, known as Dragonfly or Energetic Bear, began a campaign in late 2010 [75] with the intention of targeting the energy sector and industrial control systems (ICS) through their supply chain. In other words, the Dragonfly group attacked the suppliers of the target instead of attacking the target directly.

The Dragonfly group applied at least three different infection tactics against victims in the energy sector. The first one was an email spear-phishing campaign and is examined in this section. The other two tactics (i.e., second and third) are described in the next sections 2.2.2 and 2.2.3. However, the Dragonfly group used two main pieces of malware in its attacks. Both are Remote Access Tools (RAT) type malware enabling the attackers to access and control the compromised computers.

The favoured malware tool of the Dragonfly group was Backdoor.Oldrea, which was also known as Havex or the Energetic Bear RAT. Symantec reported that Oldrea was used in around 95% of infections. This malware acted as a back door for the attackers onto the victim's computer, enabling them to extract information and install further malware. In particular, Oldrea, gathered system information such as operating system, computer and user name, country, language, Internet adapter configuration information, available drives, default browser, running processes, desktop file list, My Documents, Internet history, program files, and root of available drives. In addition, Oldrea collected data from Outlook (address book) and ICS related software configuration files [95]. All this data was collected and written to a temporary file in an encrypted form before it was POSTed to the remote C&C (command-and-control) server controlled by the Dragonfly attackers.

Moreover, the second main malware tool used by the Dragonfly group was Trojan.Karagany. It was a back door programmed in C/C++ and used mainly for reconnaissance operations. Specifically, it was designed to download and install additional files and exfiltrate data. Moreover, it had plugin capability and its payload was approximately 72 KBs in size. Finally, Trojan.Karagany contained a small embedded DLL file, which monitored WSASend and send APIs for capturing "Basic Authentication" credentials [95].

According to the first approach (i.e., email spear-phishing campaign), selected executives and senior employees in target companies received emails with a malicious PDF attachment. Symantec states that the infected emails had two possible subject lines: "The account" and "Settlement of delivery problem". In addition, all the emails were from a single Gmail address. The email spear-phishing campaign was conducted from February 2013 to June 2013 [95].

2.2.2 Watering Hole Attacks

To attack an organization, cyber criminals "trojanize" a legitimate website often visited by the target company's employees. RSA Advanced Threat Intelligence Team correlated this behaviour with the one of a lion waiting for its prey at a watering hole, hence the name. RSA was the first to use the term "watering hole", in late July 2012 [102]. Here are a few examples of watering hole attacks from the wild.

VOHO

According to [102], "VOHO" campaign targeted Financial Services or Technology Services in Massachusetts and Washington, DC. This campaign worked by inserting JavaScript element in the legitimate website that would redirect the victim (i.e., website visitor) unknowingly to an exploit website. Then, the exploit website would check if the user was running a Windows machine and Internet Explorer browser, and then it would install a version of gh0st RAT. "gh0st RAT" was a Remote Access Trojan that allowed attackers to control the infected endpoints, log keystrokes, provide live feeds of webcam and microphone as well as download and upload files.

Dragonfly - 2nd tactic

As described before in "Dragonfly - 1st tactic" section, the Dragonfly group has used at least three infection tactics against targets in the energy sector. After the earliest tactic (i.e., email spear-phishing campaign) that was described in "Dragonfly - 1st tactic" section, the Dragonfly attackers shifted their focus to watering hole attacks. It was noticed that this shift happened in June 2013 [95]. The Dragonfly attackers compromised a number of energy-related websites and injected an iframe into each of them. Then, this iframe would redirect users to another legitimate, but also compromised, website hosting the Lightsout exploit kit, as shown in figure 2.2. This in turn would exploit either Java or Internet Explorer to download Oldrea or Karagny on the target's machine.

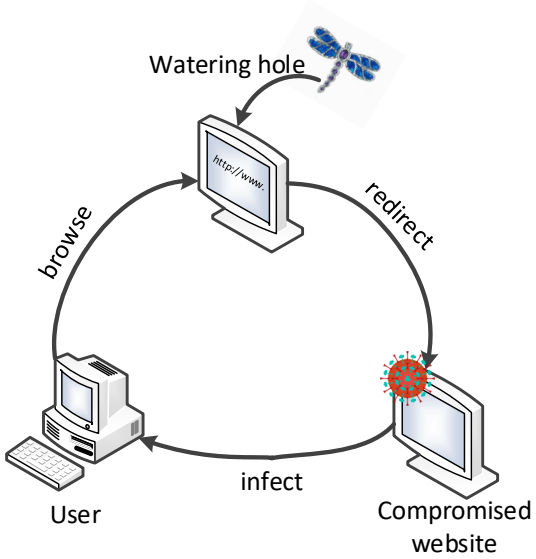

Figure 2.2: Semiconductor supply chain information flow.

Besides, in September 2013, the Dragonfly group started using a new version of this exploit kit, known as the Hello exploit kit. The main web page for this kit contained

JavaScript that was able to identify installed browser plugins. Then, the victim was redirected to a URL which in turn determined the best exploit to use according to the collected information [95].

Shylock

In November 2013, BAE Systems Applied Intelligence announced that a series of legitimate websites had been infected with the Shylock malware [5]. The cyber-criminals infected a legitimate website by inserting a JavaScript file that initially identified when the browser was used and then this JavaScript file was responsible to show a message, in the browser's style, prompting the user to download the malware that, however, was presented as innocent software. BAE Systems [6] gave the following message example: "Additional plugins are required to display all the media on this page", with a button saying "Install Missing Plugins...". In case that the user decided to proceed and install the "missing plugins", the Shylock malware was installed on his/her machine.

2.2.3 Attacks based on "trojanized" Third-party Software

This section includes a real-life example of attacks based on "trojanized" software of ICS equipment providers.

Dragonfly - 3rd tactic

The third tactic of the Dragonfly group was the infection of a number of legitimate software packages. In particular, three different ICS equipment providers were targeted and the Dragonfly attackers inserted malware into the software bundles that these providers had made available online for download from their websites [95]. The first provider discovered that it was compromised shortly after infection (time not specified), but the malware had already been downloaded 250 times. The second provider had infected software available for download for at least six weeks and the third provider had infected software available online for ten days, approximately [96].

2.2.4 Attacks based on Malicious Code and Counterfeit Certificates

This section includes a real-life example of attacks based on malicious code and counterfeit certificates in industrial environment.

Stuxnet

The German Steel Mill attack described earlier is not the first attack that caused physical damage of equipment. The first one was the Stuxnet attack [51] that was designed to target SCADA systems and was responsible for attacking an Iranian nuclear facility.

Stuxnet exploited four zero-days vulnerabilities, compromised two digital certificates, injected code into ICS and hid the code from the operator [32]. After implementing the code (process that probably took a long time), the attackers had to steal digital certificates, in order to avoid detection [32]. Stuxnet compromised the system via USB [59], [58] and infected every Windows PC it could find. However, in terms of controllers, it was much pickier. It targeted only controllers from one specific manufacturer (Siemens) [59].

2.2.5 Attacks based on Tampered Devices

This section includes a real-life example of attacks based on tampered devices in business environment.

Michaels Stores

In May 2011, Michaels Stores reported an attack that allowed criminals to steal credit and debit cards and the associated PIN codes. To steal this information, attackers tampered at least 70 point of sale (POS) terminals [53]. In a blog entry from Krebs on Security [54], Krebs explained that there are few ways to tamper with POS terminals. One way is to have pre-compromised terminals ready to be installed at the cash register. In addition, fake POS terminals can also be used to record data from swipe cards and PIN entry. For precaution, Michaels Stores replaced 7,200 PIN pads and trained employees to check regularly if the equipment had been compromised.

Chapter 3

Security Framework Protocols

In this Chapter, we provide the description of the SSL/TLS, OAuth 2.0 and OpenID Connect. Moreover, an overview of the most well-known attacks against SSL/TLS and OAuth 2.0 is also given.

3.1 Transport Layer Security

The Transport Layer Security (TLS) protocol is used to establish a connection between two parties in a secure way. TLS can be considered as version 3.1 of SSL, as it is based on SSL 3.0 Protocol [25]. The main objective of this protocol is to provide privacy and data integrity between two communicating entities over the Internet. TLS consists of two layers: the TLS Record Protocol and the TLS Handshake Protocol.

The TLS Record Protocol is at the lowest level and provides connection security that has two basic properties: a) the connection is private and b) the connection is reliable. To achieve the first property (i.e., private connection), symmetric cryptography is used for data encryption, where the keys are generated uniquely for each connection and are based on a secret, negotiated by the TLS Handshake Protocol. On the other hand, to achieve the second property (i.e., reliable connection), the message transport includes a message integrity check using a keyed MAC [25].

On top of the TLS Record Protocol, the TLS Handshake Protocol runs to allow the two communication entities to authenticate each other and to negotiate a cipher and cryptographic keys before the application protocol transmits/receives its first byte of data. In particular, the TLS Handshake Protocol provides connection security with the following three basic properties. Firstly, the communicating entities can authenticate each other by using asymmetric cryptography (e.g., RSA). Secondly, the negotiation of a shared secret is secure so that the secret will remain unavailable to an attacker (i.e., eavesdropper), and for any authenticated connection the secret cannot be revealed to the attacker; Finally, the negotiation is reliable so that an attacker will not be able to modify the negotiation without being detected by the communicating entities [25].

3.1.1 SSL/TLS Handshake Protocol

The purpose of the TLS handshake protocol is to select a cipher spec and generate a master secret. These are the primary cryptographic parameters for a secure session. It is also possible to authenticate the parties if they have trusted certificates. There are three authentication modes: authentication from both parties, authentication from server and complete anonymity [25].

The SSL/TLS handshake protocol consists of message exchange between client and server. These messages allow client and server authentication and negotiation of cryptographic keys, encryption and MAC algorithms. Figure 3.1 shows the sequence diagram of the SSL/TLS establishment procedure:

1. The client initiates a session by sending the ***Client_Hello*** message to the server. This message contains the following parameters:

   - Version: The highest version of the SSL/TLS protocol that the client can support during this session.
   - Random structure: A client-generated random structure that consists of a 32-bit timestamp and 28 bytes generated by a secure random number generator. These values are used as nonces during the key exchange in order to prevent replay attacks.
   - SessionID: A variable-length session identifier. A zero value indicates that the client prefers to establish a new connection on a new session. On the other hand, a nonzero value indicates that the client prefers to update the parameters of an existing connection or to create a new connection on the current session.
   - CipherSuite: A list of the cryptographic options supported by the client, in decreasing order of preference. Each cipher suite (i.e. element of the list) defines a key exchange method, a bulk encryption algorithm (including secret key length), a MAC algorithm and a PRF. Dierks in [25] provides details about the supported cipher suites.
   - Compression Method: A list of the compression methods supported by the client, sorted by client's preference.

2. The server will send the ***Server_Hello*** message in response to the ***Client_Hello*** message. This message includes the same parameters as the ***Client_Hello*** message. Particularly, it contains the following parameters:

   - Version: The lower of the versions of the SSL/TLS protocol suggested by the client in the ***Client_Hello*** message and the highest supported by the server.
   - Random structure: A random structure generated by the server. This structure must be generated independently from the client-generated random structure.

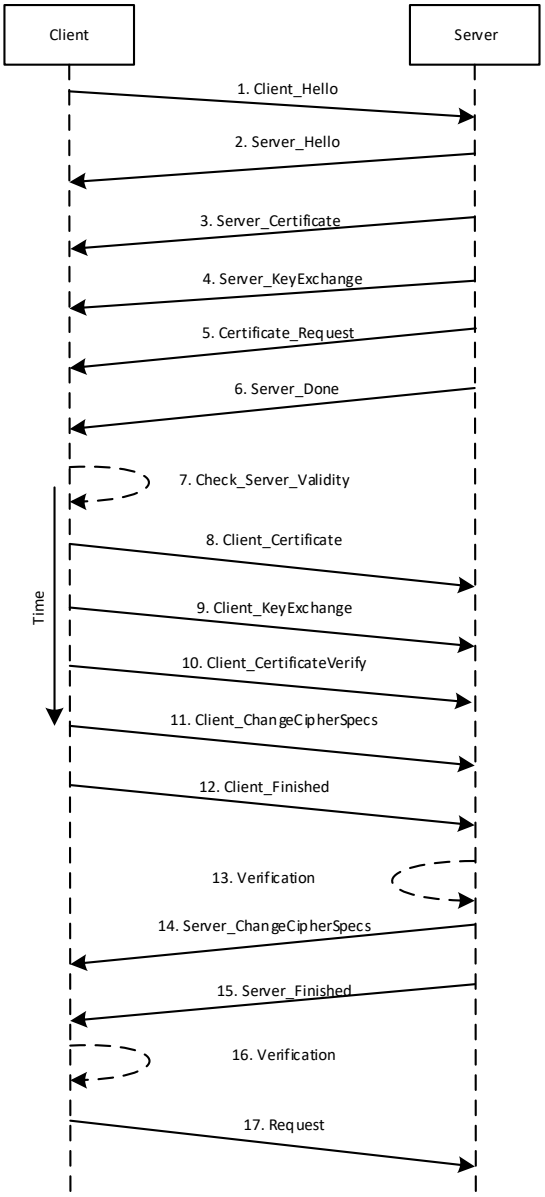

Figure 3.1: Sequence Diagram of the SSL/TLS Establishment Procedure.

- SessionID: The identity of the session corresponding to this connection. If the value of the session identifier in the ***Client_Hello*** message is nonzero, the server will look in its session cache for a match. If a match is identified and the server prefers to establish the new connection using the specified session state, then the server will respond with the same value as was provided by the client. Otherwise, the server will respond with the value for a new session.
- CipherSuit: The single cipher suite selected by the server from the list of the cipher suites provided by the client.

- Compression Method: The single compression method selected by the server from the list of the compression methods provided by the client.

3. The server sends the ***Server_Certificate*** message including the server's Public Key Certificate (PKC). This message is required for any agreed-on key exchange method apart from the anonymous Diffie-Helman key exchange method.

4. The server sends the ***Server_KeyExchange*** message. This message is sent only when the ***Server_Certificate*** message (if sent) does not contain enough data (i.e. cryptographic data) to allow the client to exchange a pre-master secret.

5. The server sends the ***Client_CertificateRequest*** message to request the PKC of the client. Consequently, the connection can be authenticated mutually.

6. The server sends the ***Server_Done*** message to indicate the end of the ***Server_Hello*** and the associated messages. Then, the server is waiting for the client's response.

7. Upon receiving the ***Server_Done*** message, the client should verify that the server provided a valid PKC and also check that the parameters of the ***Server_Hello*** message are acceptable.

8. If all checks are satisfactory, the client sends the ***Client_Certificate*** message including its PKC.

9. The client sends the ***Client_KeyExchange*** message. With this message, the pre-master secret is set. The content of this message depends on the agreed-on key exchange method.

10. The client sends the ***Client_CertificateVerify*** message to provide explicit verification of its PKC. This message is only sent following any client's certificate that has signing capability.

11. The client sends the ***Client_ChangeCipherSpecs*** message to indicate that the following messages sent by the client will be encrypted using the agreed-on security parameters.

12. The client sends the ***Client_Finished*** message to verify that the key exchange and authentication processes were completed successfully.

13. Upon receiving the ***Client_Finished*** message, the server must verify that the message's content is correct (i.e. verify the integrity of the handshake process). If the verification process fails, the connection is rejected.

14. Otherwise, the server sends the ***Server_ChangeCipherSpecs*** message to indicate that the following messages sent by the server will be encrypted using the agreed-on security parameters.

15. The server sends the ***Server_Finished*** message to verify that the key exchange and authentication processes were completed successfully.

16. Upon receiving the ***Server_Finished*** message, the client must verify that the message's content is correct (i.e. verify the integrity of the handshake process). If the verification process fails, the connection is rejected.

17. At this point, the handshake is complete and the client and the server can start exchanging application-layer data. Thus, the client sends its ***Request*** message to the server.

3.1.2 Attacks against SSL/TLS

During the past few years, we have witnessed several serious attacks on SSL/TLS protocol, including attacks on its most commonly used ciphers and modes of operation. In February 2015, the Internet Engineering Task Force (IETF) published a document summarizing known attacks on TLS and **(DTLS)!** (**(DTLS)!**) [88]. In addition, in May, IETF published recommendations to counter the attacks [87]. Most of these attacks are described below.

3.1.2.1 SSL Stripping

SSL Stripping is a form of the more generic "downgrade attack" that works by removing the SSL/TLS data from the request message. Thus, a website secured with HTTPS is downgraded to HTTP. Moxie Marlinspike developed the SSL Strip tool [71], [70] that sends the client request to the server after removing the SSL/TLS request messages, making the server think the client does not support TLS and establishing an insecure connection. A MITM attacker can achieve SSL Stripping by redirecting client requests to insecure ports and, this way the data are not encrypted. This is an exploitation of the vulnerability of having multiple ports open for the same application. It is worthwhile to mention that this attack works on websites that make use of both HTTP and HTTPS. Therefore, in order to defend against this attack, it is recommended to use only HTTPS and not HTTP. Thus, a solution to mitigate this attack is the use of HTTP Strict Transport Security (HSTS) header that commands browser to connect only through HTTPS and never HTTP.

3.1.2.2 STARTTLS Command Injection Attack

STARTTLS is an application-level command that allows upgrading a clear text connection to use TLS [44]. POP3, SMTP, FTP, XMPP, LDAP and NNTP are examples of application layer protocols that use STARTTLS [99]. The request to upgrade the connection is not protected, meaning STARTTLS is vulnerable to downgrade attacks. A MITM attacker can remove the STARTTLS indication from the request and the connection proceeds unprotected. A HSTS-based solution is necessary to mitigate this attack.

3.1.2.3 BEAST

The Browser Exploit Against SSL/TLS (BEAST) is a client-side (browser) attack that affects SSL 3.0 and TLS 1.0. This attack exploits a vulnerability in the implementation of the Cipher Block Chaining (CBC) mode which is related to the predictability of the Initialization Vector (IV) [88]. Specifically, this vulnerability allows the attacker to perform chosen plaintext attack and decrypt data exchanged between two parties [26]. To launch this attack the attacker has to implement a Man-In-The-Middle (MITM) in order to inject packets into the SSL/TLS traffic [29].

In particular, in this attack, the attacker can test if the plaintext block P_1 is equal to G (i.e., a blind guess of P_1), by observing 2 consecutive ciphertext blocks C_0 and C_1, where $C_1 = E(key, C_0 \oplus P_1)$, due to how CBC mode works. Then, the attacker selects the next plaintext block P_2 to be equal to: $P_2 = C_0 \oplus C_1 \oplus G$, and forces the victim (i.e., client) to send it to the server. Thus, the victim sends the corresponding ciphertext C_2 which is equal to: $C_2 = E(key, C_1 P_2) = E(key, C_1 \oplus C_0 \oplus C_1 \oplus G) = E(key, C_0 \oplus G)$. Now, if $C_1 == C_2$ then $G = P_1$. The BEAST attack can be mitigated by using TLS v1.1 or TLS v1.2.

3.1.2.4 Padding Oracle Attack

All versions of SSL/TLS have a MAC-Pad-encrypt design. This design was proved insecure, because it allowed padding oracle attacks [100]. In 2002 Vaudenay [100] presented an attack on CBC encryption for a specific method of padding the message. Since then, many articles prove that the attack is possible for different types of padding methods [77], [10]. Vaudenay's padding method was CBC-PAD. This method adds as many bytes as needed to the message and each byte has the value of the length of the padding. In other words, if the padding needed is 1 byte, the message will be padded with the byte 0×01, if two bytes are needed, the padding would be 0×0202, and so on. For the attack to work, it is needed an oracle O that decrypts a chosen ciphertext returning VALID if the message is properly padded, and INVALID otherwise.

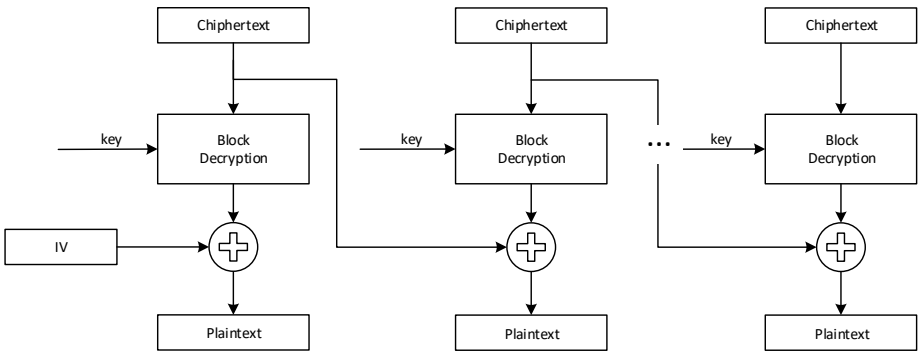

Figure 3.2: CBC Mode Decryption.

Figure 3.2 represents the CBC-mode decryption. If we consider that our message has just two blocks $C = (IV, C_1)$ of 8 bytes each, the plaintext message P_1 will have one block. Any changes in the IV will induce changes in the plaintext and it will be possible to get information about it. Vaudenay's attack starts by changing bytes in the IV until $O(C)$ = VALID. This means that the message P_1' is correctly padded and ends with 01 with probability of $1/2^8$, or 0202 with probability of $1/2^{16}$, or 030303 with probability of $1/2^{24}$, etc. Considering these probabilities, it is expected that the padding to be 01 and the last byte from P_1 is $VI \oplus 01$. However, to verify that the padding is 01 and not any of the other options (e.g.0202, 030303, etc.), we XOR the second last byte with 01. If the padding was, for example, 0202, after the XOR it would be 0302 and $O(C)$ = INVALID. After knowing the last byte of P_1, we can change the last byte of IV so that P_1 ends with 02 and try to find the correct IV until $O(C)$ = VALID. Continuing this method, we can find all bytes of P_1 [10], [100].

Black and Urtubia [10] proposed an improvement to Vaudenay's attack. First, they try to find the padding length. Instead of trying to change IV to make the plaintext end with padding 01, they change the IV from the first byte. If this byte does not belong to the padding bytes, $O(C)$ will not return an error. After this, they change the second IV byte, etc., until $O(C)$ = INVALID. This means they have reached the padding bytes. From now, the attack continues as before, the IV is changed so the padding numbers increment by one (if the padding was 0×030303, they change them to 0×040404) and proceed to alter IV until the oracle returns a valid answer.

3.1.2.5 Lucky Thirteen

After the discovery of padding attacks, the TLS protocol suffered some modifications. The padding error and the Message Authentication Code (MAC) validation error started being the same so the attacker could not understand which one had happened. However, it was still easy to figure out what error occurred, since the time needed to validate the padding bytes and evaluate MAC were very different. For this reason, now it is necessary to compute the MAC even if the padding is incorrect and consider the message as not padded. The RFC documents state that "This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal." In 2013, AlFardan and Paterson [3] proved that this statement was wrong.

This attack is possible due to how the MAC validation works. If a message has 55 or less bytes it can be encoded into a single block of 64 bytes, resulting in 2 compression function evaluations for the inner hash operation, and 2 more for the outer hash operation. Meaning that, for a 64-byte block, it is needed 4 compression function evaluations. If the message length is between 56 and 119 bytes, it takes $3 + 2$ compression function evaluation. This attack takes advantage of the extra evaluation to differentiate the computation times of these two cases.

To perform a MAC operation in a message, only the Payload is considered, meaning

that the MAC tag and padding bytes are removed. Before the Payload, 13 bytes are concatenated a sequence number of 8 bytes and a 5-byte field consisting of a 2-byte version field, 1-byte type field and a 2-byte length field (HDR). Assuming the MAC algorithm is HMAC-SHA-1, the calculated MAC tag has a length of 20 bytes. There can be three types of messages for a 64-byte plaintext message:

- The message has one byte of padding. This will result in a $64 - 20 - 1 = 43$ bytes of plaintext. Thus, the MAC verification is on a $13 + 43 = 56$ byte message.
- The message has valid padding of two or more bytes. Meaning that the MAC validation is done on a 55 or less byte message.
- The message has invalid padding, which means that MAC will be performed on a 57 byte message.

As said before, 56 or more bytes messages take longer to perform MAC validation. The attacker can exploit this timing difference to discover the last two bytes of the message.

Assuming the last plaintext block is $P = P^* \oplus \Delta$ we can send all combinations of the last two bytes of Δ to the server and study the time it takes to receive the error message (it is expected that the MAC validation fails). The faster response will be when the last two bytes form a valid padding. The messages are all sent over a multi-session attack, because the TLS session is destroyed after the first ciphertext is sent. This attack is mitigated if the CBC encryption is disabled, or by using encrypt-then-MAC instead of the TLS default of MAC-then-encrypt.

3.1.2.6 POODLE

The Padding Oracle On Downgraded Legacy Encryption (POODLE) attack is a padding oracle attack that does not rely on timing information, but still works on TLS v1.1 and TLS v1.2 as long as the SSL v3.0 downgrade is possible. The POODLE attack takes advantage of two factors. The first one is related to the fact that some servers/clients still support SSL 3.0 for interoperability and compatibility with legacy systems, and the second factor is the Block Padding-related vulnerability of SSL 3.0. In case that a client initiates the handshake phase and sends the list of the supported SSL/TLS versions to the server, a MITM attacker can impersonate the server until the client agrees to downgrade the connection to the vulnerable SSL 3.0. Then, when the connection between the client and the server is established to the vulnerable SSL 3.0 version, the attacker can benefit from the Block Padding-related vulnerability of SSL 3.0. According to RCF 7457 [88], there are no known mitigations for this attack. However, one can avoid it if SSL v3.0 is completely disabled. Finally, based on the Trustworthy Internet Movement website [79], 98.6% of the sites surveyed in December 2017 are not vulnerable to POODLE attack.

3.1.2.7 Attacks on RC4

The RC4 stream cipher is a variable-key size stream cipher that was designed in 1987 by Ron Rivest for RSA Data Security, Inc. [86]. For 7 years, this cipher was secret, accessible

only after signing a non-disclosure agreement. In 1994, the source code was made public anonymously. Since then, it has been dissected on Usenet, distributed at conference, and taught in cryptography courses. It is worth noting that in 2002 Mantin and Shamir [49] proved the non-random behaviour of this algorithm.

As explained in [86], the RC4 stream cipher has two stages. The first one is the Key Scheduling Algorithm used to initialise the permutation in an array called S-box (internal state). Its entries are a permutation of the numbers 0 through 255, and the permutation is a function of the variable-length key. To disarrange the S-box for the first time, the Algorithm 1 is used.

Input: key K of l bytes
Output: Initialization of the internal state S_0
$j = 0$;
for $i = 0 : 255$ **do**
  $j = (j + S[i] + K[i \ mod \ l]) \ mod \ 256$;
  swap $S[i]$ and $S[j]$;
end
$S_0 = (i, j, S)$;
return S_0

Algorithm 1: Key Scheduling.

Where $S[i]$ is the i-th byte of the S-box and $K[i]$ is the i-th byte of another 256-byte array with the key (repeating the key as necessary to fill the entire array).

To generate a random byte (keystream K) that will be bitwise XOR-ed with the plaintext, the Pseudo-random Generation Algorithm (PRGA) specified in Algorithm 2 is used.

Input: internal state S_r
Output: keystream byte Z_{r+1}
Output: internal state S_{r+1}
parse $(i, j, S) \leftarrow S_r$;
$i = (i + 1) \ mod \ 256$;
$j = (j + S[i]) \ mod \ 256$;
swap $S[i]$ and $S[j]$;
$Z_{r+1} \leftarrow S[S[i] + S[j]]$;
$S_{r+1} \leftarrow (i, j, S)$;
return (Z_{r+1}, S_{r+1})

Algorithm 2: Pseudo-random Generation.

In 2002, Mantin and Shamir [49] found a significant statistical weakness of RC4. However, the authors still considered the cipher to be safe, if the first 2 output words were discarded (or the first 256 words, preferably). They found out that the second output word of RC4 had a strong bias, since it has twice the probability to be zero (1/128 instead of 1/256 for n=8). In 2011, Maitra et al. [67] discovered that, after all, there were biases in the first 3 to 255 bytes of the RC4 keystream towards zero.

In February 2015, IETF released a memo called Prohibiting RC4 Cipher Suites [78]. As the name suggests, they required RC4 to be removed from the cipher suites allowed

in the TLS negotiation. Indeed, it was suggested that if the only cipher suite offered is RC4, then the handshake has to be terminated. Finally, in January 2017, the Trustworthy Internet Movement SSL Pulse [79] survey stated that only 17 of almost 140.000 surveyed websites support RC4 cipher suites.

3.1.2.8 Compression Attacks

In 2002, Kelsey [50] published a paper about how lossless compression algorithms reveal information about the plaintext being transmitted. However, this issue resurfaced in 2012 when Duong and Rizzo presented their CRIME attack at Ekoparty Security Conference [30]. Compression attacks take advantage of the algorithms that HTTP uses to compress data. GZIP is the most common used over the web protocols and it is based on the DEFLATE algorithm [24], that uses a combination of the LZ77 and Huffman coding [48].

LZ77 algorithm [104] compresses data by replacing repeated occurrences with a reference to a single copy of uncompressed data that appears earlier in the stream. The reference is a pair of numbers that represents distance and length. This algorithm has a limited window to compress data, meaning that if the repetition is out of the window, it is kept as is with no reference to earlier appearance. In Figure 3.3, an example of the LZ77 algorithm, retrieved from [103], presents how the compression works: repeated strings are replaced with a reference.

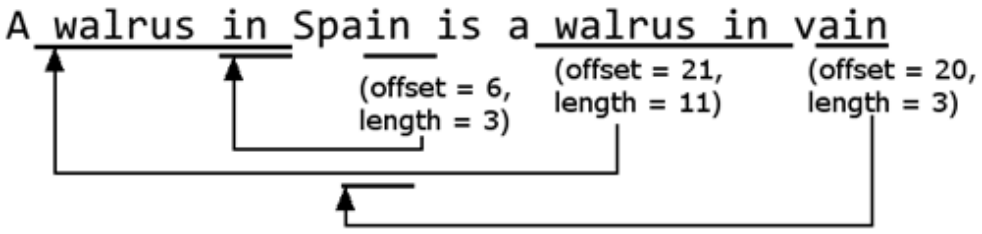

Figure 3.3: LZ77 Algorithm Example. (Source: [103])

In the following sections, we present three attacks that abuse compression in order to decrypt ciphertext. To avoid these attacks, SSL/TLS compression should be disabled.

CRIME

The Compression Ratio Info-leak Made Easy (CRIME) attack was created by Duong and Rizzo, the authors of the BEAST attack. This attack allows the attacker to steal cookies and hijack sessions by exploiting the data compression feature of the SSL/TLS protocol. Particularly, this attack benefits from the way duplicate strings are eliminated in order to obtain session tokens by brute forcing them [27], [28]. Hence, the amount of redundancy in data affects the amount of compression. It leads to the fact that more redundant data will show more compression and thus smaller length of the HTTP request. Therefore, when the length of a crafted request is smaller than the length of another crafted request, the attacker knows he/she guessed a secret byte correctly.

A request message contains a controlled URL, public headers and unknown secret cookies, from which the attacker knows only the name. Compression puts together all these parts and it becomes a problem.

Let's assume the attacker is trying to identify the secret cookie from the following message.

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=du5h3o5ja1fis830lrlsm7yji892vm4p
```

The attacker can resend the message with a different URL.

```
POST /target HTTP/1.1
Host: example.com?secretcookie=a
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=du5h3o5ja1fis830lrlsm7yji892vm4p
```

At this point, the attacker can use a network sniffer to verify message length. The atacker can then send other values for the **secretcookie**: b, c, d, etc. When the message length gets smaller, it means that a bigger sequence of characters has been duplicated.

```
POST /target HTTP/1.1
Host: example.com?secretcookie=d
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=du5h3o5ja1fis830lrlsm7yji892vm4p
```

For wrong guesses, the LZ77 algorithm will compress the duplicated **secretcookie=** string. However, for a correct guess, the recurring string will be **secretcookie=d**. Thus, the resulting request will be smaller and the attacker will realize that he/she has retrieved the first character of the secret cookie. The same process is repeated by considering the value of the secret cookie as **secretcookie=d** and the second value is to be retrieved. Similarly, the same process is repeated until the entire secret cookie value is revealed. The CRIME attack is possible in all versions of SSL/TLS and it can be mitigated by disabling SSL/TLS compression [84].

TIME

Despite the high impact of the CRIME attack on the world of security, this attack has two major practical drawbacks. Firstly, it targets only HTTP requests while most of the current web does not compress HTTP requests. Actually, the few protocols that supported HTTP requests compression have stopped their support by thus rendering this attack irrelevant [7]. Secondly, the attack model of the CRIME attack limits the attack to mostly Man-in-the-Middle (MITM) cases. This is because the attacker should not only

control the plaintext but also be able to eavesdrop the ciphertext [7]. These two limitations are addressed by the Timing Info-leak Made Easy (TIME) attack [7] that targets the web responses instead of the web requests. By changing the target of the attack, the attack surface increases, since most of the current web makes use of HTTP response compression in order to save bandwidth and latency. In addition, the TIME attack uses timing information differential analysis to infer on the compressed payload's size and thus, its attack model can be simplified. Therefore, according to TIME's attack model, the attacker only needs to control the plaintext [7]. The TIME attack can be mitigated by disabling SSL/TLS compression [88].

BREACH

The Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) attack was announced in 2013 [38]. This attack shows that TLS-protected traffic remains vulnerable to realistic compression side-channel attacks. It is achieved by attacking HTTP responses instead of HTTP requests. Even if TLS-level compression is disabled, it is very common the use of GZIP at the HTTP level. Moreover, numerous web applications include secrets, such as Cross-Site Request Forgery (CSRF) tokens, and user input, such as URL parameters, in the same HTTP response, and thus (very likely) in the same compression context. Consequently, we can say that the BREACH attack can be performed similar to the CRIME attack, but without relying on TLS-level compression. The BREACH attack proceeds byte-by-byte. Initially, the attacker forces the victim to send a small number of requests to guess the first byte of the target secret. Then, the attacker measures the size of the resulting HTTP responses and based on that information, the oracle determines the correct value for the first character of the secret [38]. Although, Gluck et al. in [38] mention that they are unaware of a clean, effective, and practical solution to the problem, they suggest the following tactics for mitigating the BREACH attack:

- **Length hiding**. Since the attack relies on measuring the length of the ciphertext, a natural mitigation approach is to hide this information from the attacker. It can be simply achieved by adding a random amount of garbage data to each response. This approach requires the attacker to issue more requests and measure the sizes of more responses, but it is not enough to make the attack infeasible. Actually, the attacker can learn fast the truth length of the ciphertext, by repeatedly sending requests and averaging the sizes of the corresponding responses.

- **Separating secrets from user input**. If these two types of information (i.e., user input, secrets) are not compressed together, this attack can be avoided. However, depending on the nature of the application and its implementation, this may be very tedious and highly impractical. Disabling HTTP-level compression. This solution obviously eradicates the attack. However, this solution leads to a rather drastic impact on performance.

- **Masking secrets**. Tom Berson [38] created a method so that the targeted secret will not remain the same between requests. Particularly, his method allows to mask a secret S by creating a one-time pad P and embedding $P||(P \oplus S)$ instead of S, in the page. This method ensures that the secret will not be compressible.

- **Request Rate-Limiting and Monitoring**. Although the attack does not need an impossible amount of requests, it does need more requests in a short amount of time than a human user could perform. Thus, by monitoring the volume of traffic per user, the attack can at least be slowed down significantly.

- **More Aggressive CSRF Protection**. One way to thwart the attack is to require a valid CSRF token for all requests that reflect user input, since the attack will make many wrong guesses before retrieving the correct token.

3.1.2.9 Certificate and RSA-Related Attacks

Over the years, there have been multiple practical attacks on TLS when it is used with RSA certificates [20]. Some of the most well-known RSA certificate-related attacks against TLS are presented below.

Bleichenbacher Attack

Bleichenbacher [11] took advantage of Davida's findings [20] and the fixed structure of RSA to create an attack that would decrypt the PreMasterSecret. This master secret is all what an attacker needs to derive all the particular session keys [52], meaning that the TLS session is no longer secure. For this attack to be viable, it is necessary the access to an oracle that, for each ciphertext, returns an identifiable error if the plaintext is not PKCS conforming (i.e., PKCS#1).

This is an adaptive chosen-ciphertext attack, because the attacker chooses the ciphertext that he/she sends to the oracle and the next ciphertext depends on the answer from the oracle. The attacker gains information on the plaintext gradually. A detailed description is provided in [11]. Bleichenbacher states that 2^{20} chosen ciphertexts should be sufficient to derive the message sent. However, this number depends on the implementation details and thus the number of the chosen ciphertexts can vary. Version 1.0 of TLS mitigates this attack by instructing security architects not to give out information about the structure of plaintext [52].

Klima Attack

In 2003, Klima [52] presented an extension of Bleichenbacher's attack on PKCS#1. To prevent Bleichenbacher's attack, the server does not reply to a not PKCS conforming message with an error. Instead, the server generates a new `PreMasterSecret`. The communication will break down after the client's Finished message. However, the attacker does not know if the error is due to wrong `PreMasterSecret` or incorrect message format.

Another countermeasure for Bleichenbacher's attack is to verify the correctness of a version number. Version 1.0 of TLS tells security architects to validate version number, but does not tell them how. Thus, Klima realized that some implementations would issue a specific error message and that it "opened up a Pandora's box". This error message created a new side channel attack as a Bad-Version Oracle.

In version 1.1 of TLS [81], the `PreMasterSecret` version number has to be equal to the one offered by the client, not the version negotiated for the connection, in order to prevent rollback attacks. Clients have to check the version number to be sure it is correct. However, servers can check if they want. If the version number is incorrect, the server has to randomize the `PreMasterSecret` instead of generating an error. By telling security architects specifically not to generate an error, both the Bleichenbacher's attack and the Klima's attacks are mitigated.

Brumley

In 2003, Brumley and Boneh challenged the assumption that common implementations of RSA (using Chinese Remainder and Montgomery reductions) were not vulnerable to timing attacks [13]. The attack was aimed at OpenSSL since it was the most commonly used in web servers. However, despite the fact that OpenSSL had a defence mechanism against timing attack, it was turned off by default.

RSA's ciphertext decryption is in the form of $m = c^d \ mod \ N$, where c is the ciphertext, d is the private decryption exponent and $N = pq$ is the RSA modulus. Brumley and Boneh's timing attack allows the discovery of the factors of N. After knowing p and q, the attacker can compute the decryption key by computing exponent $d = e^{-1} \ mod \ (p-1)(q-1)$. The attack targets q, the smaller factor. After q is known, the RSA modulus is factored and the server's private key is revealed.

To initialize the attack, one chooses a guess for q and then time the decryption of all possible combinations of the top 2 or 3 bits. When a graph is plotted, it shows two peaks, one for q and one for p. The first peak is related to the smaller value of the two, which is the q. The authors tested the attack and stated that a typical attack takes about 2 hours to complete. After these timing findings, several crypto libraries, such as OpenSSL included, started to implement blinding by default [13].

3.1.2.10 Certificate Fuzzing Tool

When implementing TLS, one of the steps is certificate validation. The main difficulty of this step is to test the correctness of the certificate validation logic in SSL/TLS implementations. It is quite difficult to create enough test inputs and interpret the result of testing. Brubaker et al. [12] realized this difficulty and created a Fuzzing Tool to systematically test SSL/TLS implementations regarding certificate validation logic. They found that the tool is able to uncover numerous vulnerabilities in popular TLS implementations related to certificate validation. One of their findings was that some browsers, when confronted with an expired and self-signed certificate, would report to the user that

the certificate was expired, but would not mention the fact that it was also invalid. Users usually ignore this report and click through the warning [1].

3.1.2.11 Man-in-the-Middle (MITM) Attacks

It is possible to compromise a SSL/TLS-based communication if an attacker positions himself in the middle of the communication channel between the client and the server. The attack is performed by creating two SSL/TLS-based connections: one with the client; and one with the server. This way, the attacker can rely messages exchanged between the client and the server without any of them being aware of the attacker's presence. To achieve this type of attacks, the attacker can exploit forged SSL/TLS certificates that can be illegitimately generated by the attacker himself or obtained by compromising trusted CAs [47], [19]. Afterwards, these certificates are used by the attacker to impersonate as the server to the client. The attack to DigiNotar [4] is an example of compromising trusted CAs in the real world.

Figure 3.4 shows the sequence diagram of this attack. It demonstrates the interactions between the attacker and the client along with the communication between the attacker and the server in the case of a potential MITM attack against the SSL/TLS-based communication. In this case, the attacker intends to impersonate the server by using a forged certificate and the client is the victim. Thus, the following steps take place:

1. The attacker positions himself into the communication channel between the client and the server, and intercepts the *Client_Hello* message (**message 1**) that the client sends to establish a new SSL/TLS connection with the server.

2. The attacker responds to the *Client_Hello* message using a forged certificate (**messages 4, 6, 8, and 11**).

3. If the client verifies the forged certificate (**message 12**), then the client establishes the SSL/TLS connection with the attacker (**messages 13, 15, 17, 19, 21, 22, 26, 29, and 30**).

4. In parallel to the previous steps, the attacker establishes a separate SSL/TLS connection with the server, impersonating the client (**messages 2, 3, 5, 7, 9, 10, 14, 16, 18, 20, 23, 24, 25, 27, and 28**).

5. At this point, the attacker has established two separate SSL/TLS-based connections: a) one with the client and b) one with the server. Now, the attacker is able to decrypt, re-encrypt and forward all exchanged messages between the client and the server (**messages 31, 32**).

There are two possible ways to mitigate this kind of attack. One solution is that the CAs should protect their critical infrastructures not only with cryptographic security mechanisms but also with intrusion detection mechanisms [72]. In addition, another solution is the use of secondary channels, such as Tor, so that the client will get additional copies of

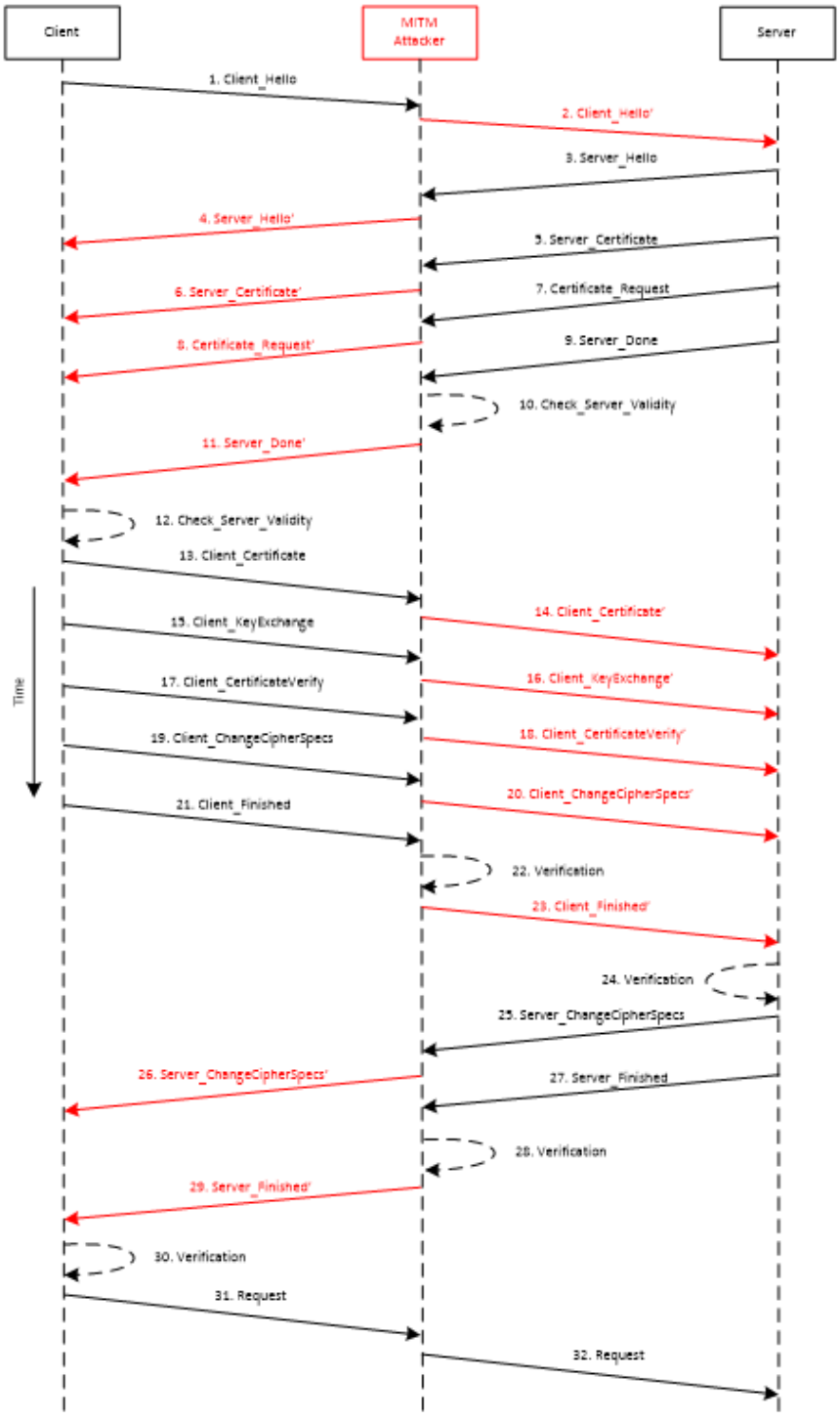

Figure 3.4: A MITM Attack against the SSL/TLS-based Communication between the Client and the Server.

the server's certificate. Assuming the attacker has no control over the secondary channels, any inconsistency in the certificates will indicate a possible MITM attack [19].

3.1.2.12 Virtual Host Confusion

In [22] and [23] Delignat-Lavaud and Bhargavan describe a new class of vulnerabilities in TLS when used in a cloud environment. The virtual host confusion attack is an attack where an attacker can take advantage of SSL 3.0 fallback and improper handling of session caches on the server side in order to establish a malicious connection to a virtual host other than the one initially intended and approved by the server [22]. In particular, the virtual host confusion attacks are very serious in performance critical environments where sharing of SSL 3.0 session caches is very common. Delignat-Lavaud and Bhargavan noted that this vulnerability has been present in Akamai servers for almost 15 years without getting noticed [23].

3.1.2.13 Computational Denial of Service (DoS) Attacks

SSL/TLS-based communication is vulnerable to computational Denial of Service (DoS) attacks [25], [72], [16], [69]. This type of attack exploits the fact that the SSL/TLS protocol handshake process is computationally expensive, and that this process is more resource intensive to the server's side compared to the client's side. Specifically, the SSL/TLS handshake requires 10-15 times more processing power on the server than on the client [16], [76]. Thus, an attacker can pretend to be a legitimate client and launch a computational DoS attack against the server by initiating many SSL/TLS handshakes so that the server will be forced to do a lot of computations. This additional workload depletes the server's CPU resources leading the server to be unavailable to the legitimate clients (i.e. denial of service). Indeed, there are two variations of computational DoS attacks: a) the computational DoS attacks based on multiple connections, and b) the computational DoS attacks based on Renegotiation [72].

In the first variation of computational DoS attacks, multiple attacking entities send fake data to the target server that supports the SSL/TLS protocol as they try to establish a number of junk SSL/TLS connections with that server. This generates extra workload to the server that exhausts its resources, because the target server believes these connections are legitimate SSL/TLS handshake attempts [63], [65].

On the other hand, in the second variation of computational DoS attacks, a single malicious host launches an attack against the targeting server. Specifically, when the SSL/TLS Renegotiation feature is enabled on the server, an attacker using only one malicious host can send multiple Renegotiation requests that will exhaust the server's resources, due to the fact that each Renegotiation request initiates a new SSL/TLS handshake which requires at least 10 times more processing power for the server than the client [76]. Hence, this kind of attack causes denial of service on the server's SSL/TLS interface [76].

Hence, the SSL/TLS-based communication between the client and the server can be vulnerable to both the computational DoS attacks based on multiple connections and

based on Renegotiation. However, there are countermeasures to prevent or minimize the effectiveness of these attacks against the SSL/TLS-based communication between the client and the server. First of all, the server can mitigate these attacks by limiting the incoming rate of new SSL/TLS connections and renegotiations [76]. Another mitigation technique can be the use of a SSL Accelerator, which is a hardware accelerator, typically a separate card that plugs into a PCI slot that offloads processing of the computationally expensive SSL/TLS handshake on the server's side [76], [36]. Finally, client puzzles can be used to mitigate these two types of attacks, since solving client puzzles increases the workload on the client's side [21], [8].

3.1.2.14 Implementation Issues

Although the Transport Layer Security protocol is well documented and properly specified, it does not mean that implementation errors affecting the level of the provided security are avoided. Indeed, several implementation errors were discovered in the past. For example, some implementations were not validating the server's identity [88]. This validation amounts to matching the protocol-level server name with the certificate's Subject Alternative Name field. However, this same information is often also found in the Common Name part of the Distinguished Name, and thus, some validators retrieve it from there incorrectly instead of the Subject Alternative Name [88]. Another implementation issues noticed by the RFC 7457 [88] is that the certificate chain is not validated at all, or the validation of the certificate chain is performed incorrectly. Some other examples of TLS vulnerabilities due to implementation issues are the following:

Python-Bugzilla

In 2013, it was found that the Python library that is responsible for the interaction with Bugzilla was not performing server's certificate validation when communicating over HTTPS [101]. Not validating Bugzilla's server certificate leaves the connection vulnerable to MITM attacks.

Heartbleed

OpenSSL introduced a coding mistake when implementing the TLS/DTLS extension Heartbeat. This allowed an exploit known as Heartbleed. The Heartbeat extension was motivated by the need for session management in Datagram TLS (DTLS) [31]. Heartbeat allows each party of a TLS connection to verify whether its peer is still present. Peers indicate support for the Heartbeat extension during the initial TLS handshake. Following negotiation, each peer sends a HeartbeatRequest message to verify connectivity. This message consists of a 1-byte type field, a 2-byte payload length field, a payload, and at least 16 bytes of random padding. Upon receiving the request, the receiver responds with a similar HeartbeatResponse message in which it echoes back the HeartbeatRequest payload and its own random padding. OpenSSL implemented this extension in a way that the

receiver trusts the request length field for the payload and if the request has a length larger than the payload, the response will reveal private information[31].

3.1.2.15 Usability

When a browser detects security issues regarding the certificate (e.g., expired certificate or self-signed certificate), it allows the user to choose whether they want to proceed to the website or not. Usability studies have shown that a large number of users ignore the warnings provided by the browsers about certificates [85], [94], [2], [33]. Hence, these users are vulnerable to interception attacks using self-signed certificates [47]. The main problem with this warning is that it always shows the same message and has the same presentation. As a result, the average user will get used to it and start to ignore it. To prevent this, more informative UI design is required so that the message shown to the user will stop being generic and provide some information about the warning reasons: expired certificate, unknown Root CA, etc.

3.2 OAuth 2.0

Nowadays, the need for secure authentication and authorization across companies has been significantly important, and according to [15], the most widely adopted protocol in this domain is OAuth. Indeed, OAuth is currently deployed by many major companies, such as Facebook, Google and Microsoft. Initially, OAuth was designed to provide a secure authorization mechanism for websites, as it defines a process for end-users to authorize third-party websites to access their private resources hosted on a Service Provider (SP) on their own behalf. Nonetheless, ever since OAuth was successfully adopted by the industry, major identity providers (e.g., Facebook, Google, and Microsoft) have used it for user authentication as well [15]. Namely, the OAuth protocol enables a user to prove his/her identity to a relying party (i.e., website) by utilizing his/her existing session with the SP [15], [93], [92]. Furthermore, the developers' community has re-targeted OAuth to mobile platforms, in addition to the traditional web platform, and thus, OAuth has become today a de-facto authentication and authorization protocol for mobile apps [15]. The latest version of OAuth protocol is the OAuth 2.0 protocol [42] that has obsoleted the OAuth 1.0 protocol described in [41]. According to [42], the OAuth 2.0 protocol defines four different roles:

a. Resource Owner (i.e., User): an entity capable of granting access to its protected resources,

b. Resource Server: the server hosting the Resource Owner's protected resources,

c. Client (i.e., Relying Party (RP)): an application requesting access to the protected resources on behalf of the Resource Owner, and

d. Authorization Server: the server that issues access tokens to the Client after successfully authenticating the User and obtaining authorization.

According to the OAuth 2.0 protocol [42], the Client requests access to the Resource Owner's protected resources and is issued a different set of credentials than those of the Resource Owner. Specifically, the Client is issued an access token (i.e., credentials) by the Authorization Server with the approval (i.e., authorization) of the Resource Owner. The access token is used by the Client to access the Resource Owner's protected resources. The authorization of the Resource Owner is expressed in the form of an authorization grant that the Client uses to request the access token from the Authorization Server. OAuth 2.0 provides four different methods so that the Client can obtain the access token. These methods are referred to as grant types and the OAuth 2.0 documentation defines the following [42]: a) authorization code, b) implicit, c) resource owner password credentials, and d) client credentials. However, as it is mentioned in [15], only two types out of the four have been widely used in practice for native applications (i.e., desktop applications, mobile apps). These grant types are the implicit grant type and the authorization code grant type. Finally, it is worthwhile to mention that both the implicit grant type and the authorization code grant type are redirection-based flow, which means that the Client

should be capable of interacting with the Resource Owner's user agent (e.g., web browser) and capable of receiving incoming requests, via redirection, from the Authorization Server.

3.2.1 Implicit Grant

The implicit grant is the shortest flow that OAuth 2.0 provides. This type is used to obtain access tokens and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using scripting language (e.g., JavaScript). Due to the fact that the implicit grant type is a redirection-based flow, the client must be capable of interacting with the Resource owner's user agent (e.g., web browser) and capable of receiving incoming requests, via redirection, from the Authorization Server [42], [34]. As illustrated in Figure 3.5, the flow for the implicit grant consists of the following steps:

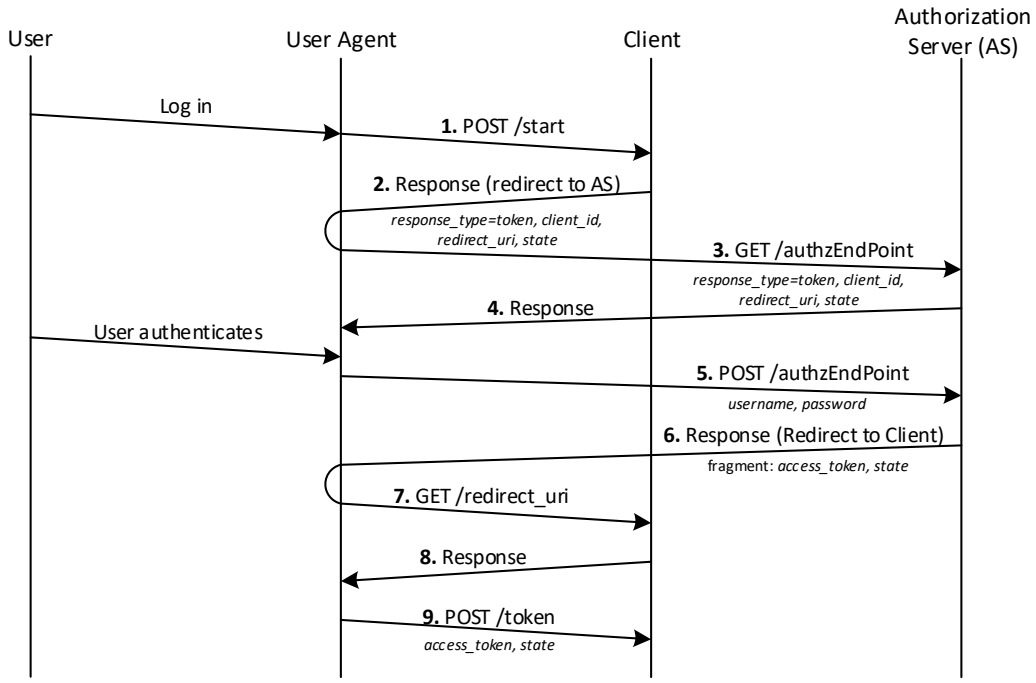

Figure 3.5: OAuth 2.0 Implicit Grant.

Step 1. The User clicks on the login button that triggers a request from the user agent (e.g. browser) to the Client (Message 1).

Step 2. The Client redirects the user agent to the Authorization Server (Message 2). The message carries the parameter *response_type* with the value 'token' to express the desire to use implicit grant, the client's identifier (*client_id*), the requested scope, a value state and the redirection URI (*redirect_uri*) to which the Authorization Server

will send the user agent back once access is granted (or denied) in Message 7. The user agent requests the authentication web page (Message 3) provided by the authorization endpoint through the GET response containing the log in form (Message 4).

Step 3. The User provides his/her username and password. The user agent sends this information to the authorization endpoint (Message 5).

Step 4. Provided that the credentials are correct and the User grants the Client's access request, the Authorization Server sends the access token (*access_token*) to the user agent, as well as the value state, provided earlier in Message 2 (Message 6). This response message redirects the user agent back to the Client using the redirection URI provided earlier in Message 2. The *access_token* and state values are sent appended to the *request_uri* as a fragment[1]. The user agent makes a GET request for the web page in *redirect_uri*. The contents of the fragment are not sent in this request (Message 7).

Step 5. The Client responds to the GET request with the contents of the web page (Message 8). With this, the Client also sends a JavaScript code that will be used to retrieve the values sent in Message 6.

Step 6. The user agent executes the JavaScript code to retrieve the *access_token* and state values and sends them to the Client (Message 9). The Client validates the state value that should be the same as the one received in message 2.

Step 7. The *access_token* can now be used to get the user's protected resources.

3.2.2 Authorization Code Grant

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Similar to the implicit grant type, it is also a redirection-based flow and thus, the Client must be capable of interacting with the Resource Owner's user agent (e.g., web browser) and capable of receiving incoming requests, via redirection, from the Authorization Server [42], [34]. As illustrated in Figure 3.6, the flow for the authorization code grant type consists of the following steps:

Step 1. The User clicks on the login button that triggers a request from the user agent (e.g. browser) to the Client (Message 1).

Step 2. The Client redirects the user agent to the Authorization Server (Message 2). The message carries the parameter *response_type* with the value 'code' to express the desire to use the authorization code grant, the client's identifier (*client_id*), requested scope, local state and a redirection URI (*redirect_uri*) to which the Authorization Server will send the user agent back once access is granted (or denied) in Message

[1] A fragment of a URI is the part sent after the '#' symbol.

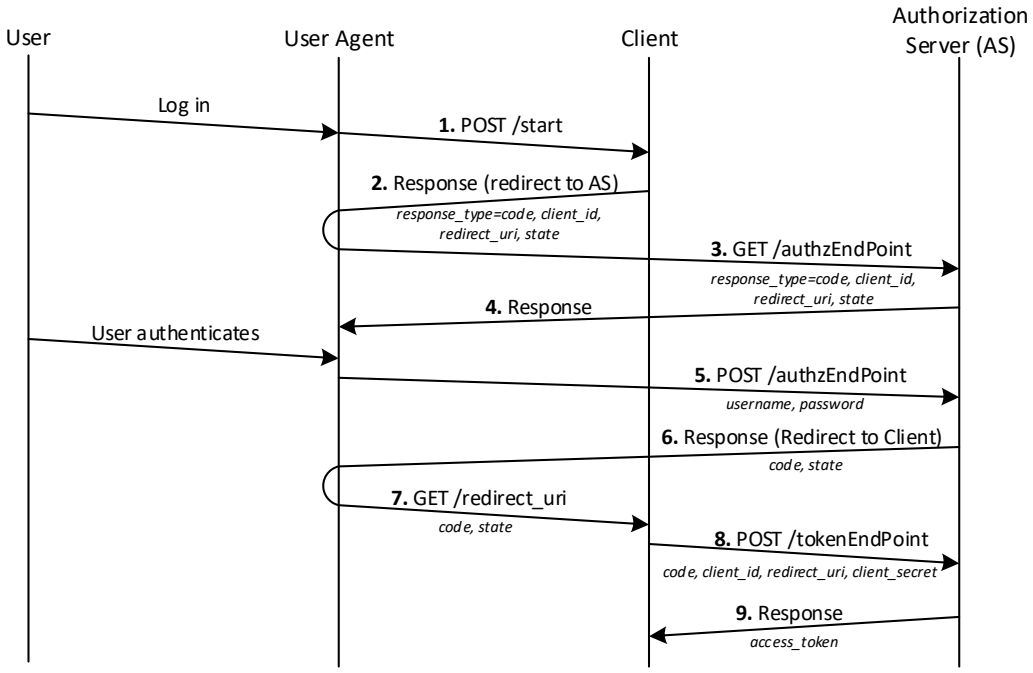

Figure 3.6: OAuth 2.0 Authorization Code Grant.

---

7. The user agent requests the authentication web page (Message 3) provided by the authorization endpoint through the GET response containing the log in form (Message 4).

Step 3. The User provides his/her username and password. The user agent sends this information to the authorization endpoint (Message 5).

Step 4. Provided that the credentials are correct and the User grants the Client's access request, the Authorization Server sends an authorization code and the value state to the user agent (Message 6). This response message redirects the user agent back to the Client using the redirection URI provided earlier in Message 2. The redirection URI includes the authorization *code* and the local *state* provided by the Client earlier (Message 7).

Step 5. The Client requests an access token from the Authorization Server by including the authorization code along with the *client_id*, *client_secret* and the *redirect_uri* (Message 8).

Step 6. The Authorization Server checks the validity of the values sent by the Client and authenticates it. In particular, it checks if the code that was issued to the Client was identified by *client_id*, and that it has not been redeemed before. Also, the Authorization Server checks if the *client_secret* is correct for the *client_id*. The value

for *redirect_uri* is also checked. If all these are valid, the Authorization Server issues an *access_token* and sends it to the Client (Message 9).

3.2.3 OAuth Vulnerabilities

In this section, we consider representative examples of known vulnerabilities of the OAuth 2.0 protocol, according to [42] and [34].

3.2.3.1 Misuse of Access Token to Impersonate Resource Owner in Implicit Flow

The implicit flow does not authenticate the Client with the Authorization Server, and since the OAuth 2.0 specification does not associate a specific access token to the requesting Client, there is no way to know if the authorization was given to the Client sending the access token to request for the user's protected information.

If an attacker obtains the access token of a legitimate user in some way (e.g., phishing), the attacker is able to impersonate the user by sending the access token to a legitimate Client. The service provider will not know that the access token was issued to the attacker's malicious Client. This attack exposes the user's protected resources and may allow the attacker to perform operations at the legitimate Client as if the attacker is the user [42].

3.2.3.2 Clickjacking

Clickjacking [42] involves tricking the user to grant access to an attacker without the user's knowledge. The attacker achieves this by registering a legitimate Client and then builds a malicious site where it loads the authorization web page of the Authorization Server in a transparent iframe overlaid on top of a set of fake buttons, which are carefully constructed to be placed directly under important buttons on the authorization page. Thus, when the user clicks a misleading visible button, he/she is actually clicking an invisible button on the authorization page (e.g., "Authorize" button). Clickjacking is a serious attack and "can cause severe damages, including compromising a user's private webcam, email or other private data, and web surfing anonymity", according to Huang et al. [46].

3.2.3.3 Stealing User Credentials

When a mobile Client (i.e., mobile app) uses embedded web browser the user is exposed, because the app has full control of the embedded browser and thus it is allowed to inject JavaScript and steals the user credentials [89].

For example, a malicious mobile app can communicate directly with the embedded browser, and thus, an attack is possible by JavaScript injection that sends the user credentials (i.e., user name and password) to the hosting malicious app as soon as the user clicks on the submit button.

3.2.3.4 Modifying the Authorization Interface

This attack [89] results from the same problem as the one before: the usage of embedded browser and the fact that the Client can communicate with it seamlessly. However, instead of stealing user credentials, this attack tries to deceive the user by showing a different list of authorizations. Thus, when the user tries to login to a malicious website with third party service provider, the Client modifies the authorization page to show different items. It can show basic info and email address, for example, instead of a list of information that the Client is actually requiring. If or when the user grants access to the Client, the user is granting access to the entire list, not shown on the page.

3.2.3.5 Attacks against System Native Browser

One way to abuse the mobile native browser is to exploit the intent manager [89]. The intent manager lets the user choose what application to launch when there are more than one to perform the specific task. A malicious app can register similar intent filters as the legitimate app, and when the user authenticates, the user is prompted to choose the app to launch. If the user selects the malicious app, the access token is sent to the attacker. At this moment, the malicious app can impersonate the legitimate application on the resource server.

3.2.3.6 Cross Site Request Forgery Attacks

A Cross-Site Request Forgery (CSRF) attack takes place when a malicious website initiates a request to a target website (e.g., by means of a HTML <img> tag). If the user browsing is authenticated to the target website, this tag will make the browser send cookies containing the user's tokens. When the malicious website sends a request to the target website, the legitimate website reacts as if the user sent the request.

To avoid this attack, the OAuth 2.0 specification recommends that the website incorporates a non-guessable state parameter and verifies if the request state is the same as the authenticated user. In [64], it was observed that not all Clients check this parameter. Some even omit the state or set it to a fixed value. These Clients are vulnerable to CSRF attacks.

3.2.3.7 HTTP 307 Redirect

In [34], Fett et al. performed an extensive analysis of the OAuth 2.0 standard and discovered two more vulnerabilities. The first one is related to the redirect code used when redirecting the user back to the Client's URI.

This attack is launched as follows: when a user is attempting to log in at a malicious client, then he/she is redirected to the Authorization Server to insert the user's credentials. The credentials are sent to the Authorization Server in a POST request. The Authorization Server checks for the correctness of the credentials and redirects the user's browser to the Client's redirection endpoint in the response to the POST request. As the 307 status code

is used for this redirection, the user's browser will send a POST request to the Client that contains all form data from the previous request, including the user credentials. To prevent this attack, the Authorization Server should always use the 303 status code, as it is the only that is unambiguously defined to drop the body of an HTTP POST request [35].

3.2.3.8 IdP Mix-Up

In this attack, the attacker confuses a Client about which Authorization Server the user chose at the beginning of the login/authorization process in order to obtain an access token which can be used to impersonate the user or access user data [34].

In this attack, the attacker manipulates the first request of the user so that the Client considers that the user want to use an Authorization Server controlled by the attacker, while the user wants to use a legitimate Authorization Server. Thus, the Client sends the authorization code or the access token issued by the legitimate Authorization Server to the attacker. Consequently, the attacker can use this information to login at the Client under the user's identity, managed by the legitimate Authorization Server, or access the user's protected resources.

To prevent this attack, the authors of "A Comprehensive Formal Security Analysis of OAuth 2.0" [34] propose that the Client should provide a different redirection endpoint for each Authorization Server. When the user is redirected back to the Client, the Client knows if the URI used is the one expected

3.2.4 OpenID Connect

OpenID Connect (OIDC) [94] was established as a standard in 2015 and it was developed to create an identity layer on top of OAuth 2.0 [4]. OIDC adds authentication to the authorization provided by OAuth 2.0, since OAuth 2.0 was not created to provide authentication and its use for authentication causes severe security flaws [95], [96]. The identity layer on top of OAuth is abstracted into a parameter called ID Token. The core parties involved in the OpenID Connect protocol as well as their relationship, as shown in Figure 3.7, are the following:

- End-user: wants to access services at the Client (i.e., Relying Party). The end-user is represented by his/her user agent. The end-user grants access to the OpenID Provider to provide user information to the Relying Party;
- User agent (UA): is typically a browser;
- OpenID Provider (OP): stores the end-user information and is capable of authenticating the end-user;
- Client (Relying Party (RP)): provides services to the end-user that require authentication.

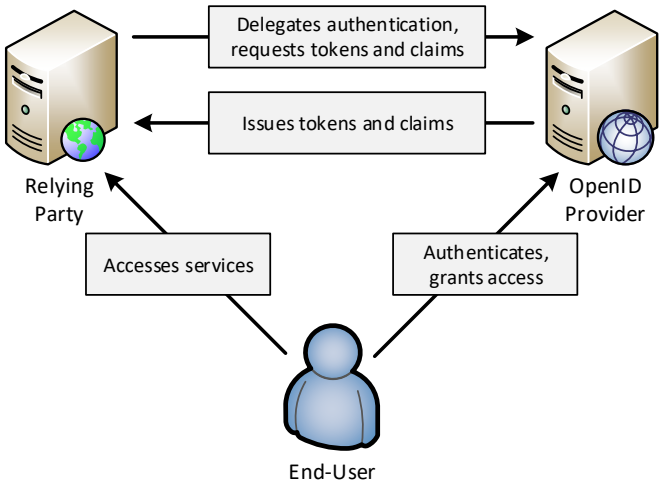

Figure 3.7: Role Relationship in OpenID Connect.

There are four endpoints defined in the OpenID Connect Core specification [94], three of them used for the flows and one for Relying Party registry:

- Authorization Endpoint: is used to perform the authentication of the end-user. In particular, it is done by sending the user agent to the Authorization Endpoint of the Authorization Server for authentication and authorization, using request parameters defined by OAuth 2 and additional parameters and parameter values defined by OpenID Connect [83].

- Token Endpoint: is where the RP communicates to obtain the tokens (*id_token* used to authenticate the end-user and *access_token* for access to the user's protected resources). Depending on the chosen OAuth 2.0 flow, the access token may or may not be sent to RP [83].

- UserInfo Endpoint: is where the RP retrieves the user's protected resources by sending the access token. These resources are information such as e-mail, address, gender, etc. [83].

- Registration Endpoint: is actually the first endpoint used. It is where the Client registers with the OpenID Provider in order to use the OpenID Connect services for authentication. When registering, the Client gives a redirection URI to OP. The OP responds with a Client ID and a Client secret that Client must store to use them in the authentication flows. The registration process from the Client to the OpenID Provider is usually done once manually or automatically with an OpenID Connect extension called Dynamic Client Registration [82].

OpenID Connect provides three authentication flows: Authorization Code Flow, Implicit Flow and Hybrid Flow. The flows determine how the Relying Party receives the ID token and access token. Next, we describe the two most used flows.

3.2.4.1 Implicit Flow

In the implicit flow, all tokens are retrieved from the authorization endpoint and it does not perform authentication of the Relying Party (RP) in the OpenID Provider. The OIDC documentation [83] states that this flow is used mostly by RPs implemented in browsers, using a scripting language. As illustrated in Figure 3.8, the implicit flow consists of the following steps [83].

Step 1. The flow begins when the User clicks on the login button that triggers a request from the user agent (e.g. browser) to the RP (Message 1).

Step 2. The RP redirects the user agent to the OpenID Provider chosen by the user (Message 2 and Message 3). This message carries the following parameters: *response_type* with the value "*id_token*" or "*id_token token*" (since the protected resources are obtained with *id_token*, the *response_type* = "*token*" is not accepted in OpenID Connect); *redirect_uri* to be used in Messages 6 and 7; *client_id*, the RP identifier; *scope* with at least the value '*openid*'; *nonce*, to mitigate replay attacks; and a recommended parameter *state*. Contrary to OAuth 2.0, the *scope* value is mandatory. In addition, the user agent requests the authentication web page (Message 3) provided by the OpenID Provider through the GET response containing the log in form (Message 4). However, the OpenID Provider must validate the request before answering (Message 4). For example, the scope parameter has to be in the request and it has to contain the value '*openid*'.

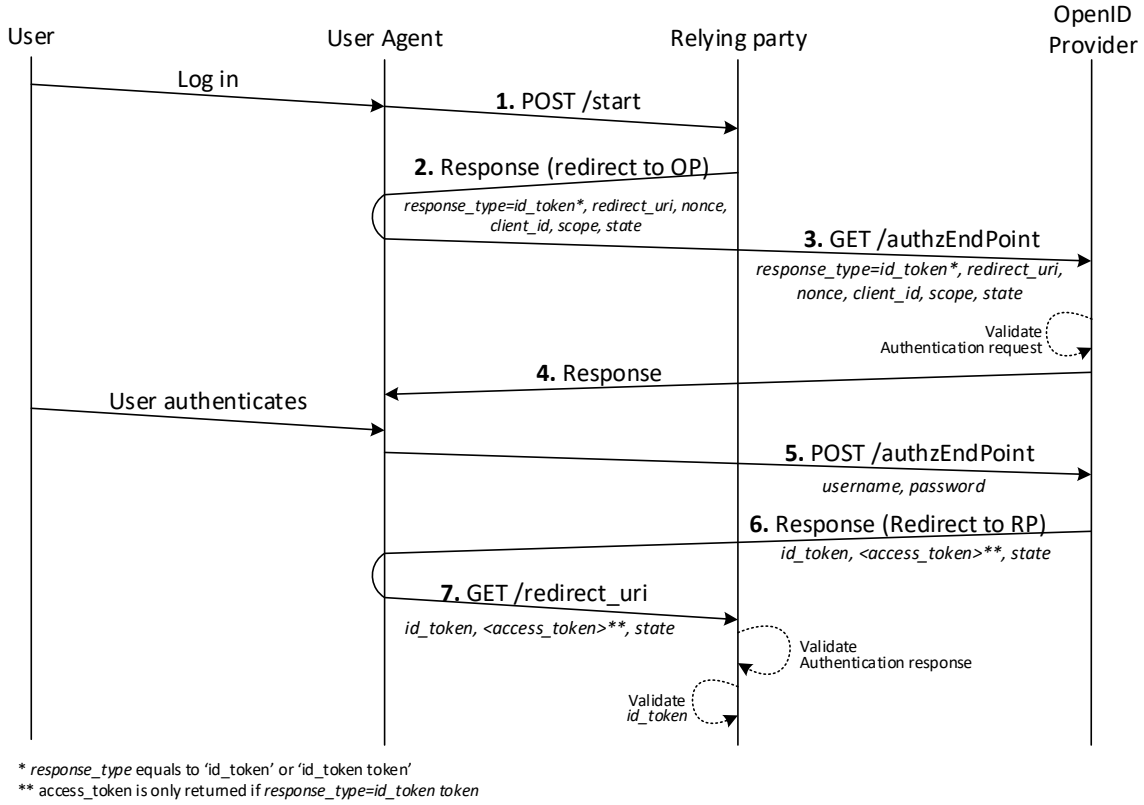

Figure 3.8: OpenID Connect Implicit Flow.

Step 3. The User provides his/her username and password. The user agent sends this information to the authorization endpoint (Message 5).

Step 4. If the user's credentials are correct and the User grants the RP's access request, the OpenID Provider redirects the user agent (Message 6) to the *redirect_uri* with the parameters: *id_token*; *access_token*, if *response_type* was "*id_token token*"; *token_type*, usually 'bearer', but it could be any value negotiated between the OpenID Provider and RP; and *state*. The latter is only present if sent in Message 3. These parameters are sent in the fragment component of the *redirect_uri* (Message 7). The RP validates the message received and all the parameters received from Message 7. If the parameter *state* was sent in Message 3, it should also be present in this message, with the same value. All unrecognised parameters have to be ignored. In addition, the parameter *id_token* is validated. If *access_token* was requested, it should also be validated.

Step 5. If the access token was requested too, and the *id_token* is valid, then, the RP can request the user's protected resources. To get the resources, the RP sends a GET request to resource endpoint with the *access_token* parameter.

3.2.4.2 Authorization Code Flow

In this section, we describe the Authorization Code Flow. In this flow, all tokens are retrieved from the Token Endpoint. Before being able to receive them, the Relying Party needs a code that later is exchanged for the tokens. This flow is appropriate for Relying Parties that can maintain a secret between them and the OpenID Provider. In other words, they can keep a secret from the user agent (e.g., browser).

As illustrated in Figure 3.9, the authorization code flow consists of the following steps [83]:

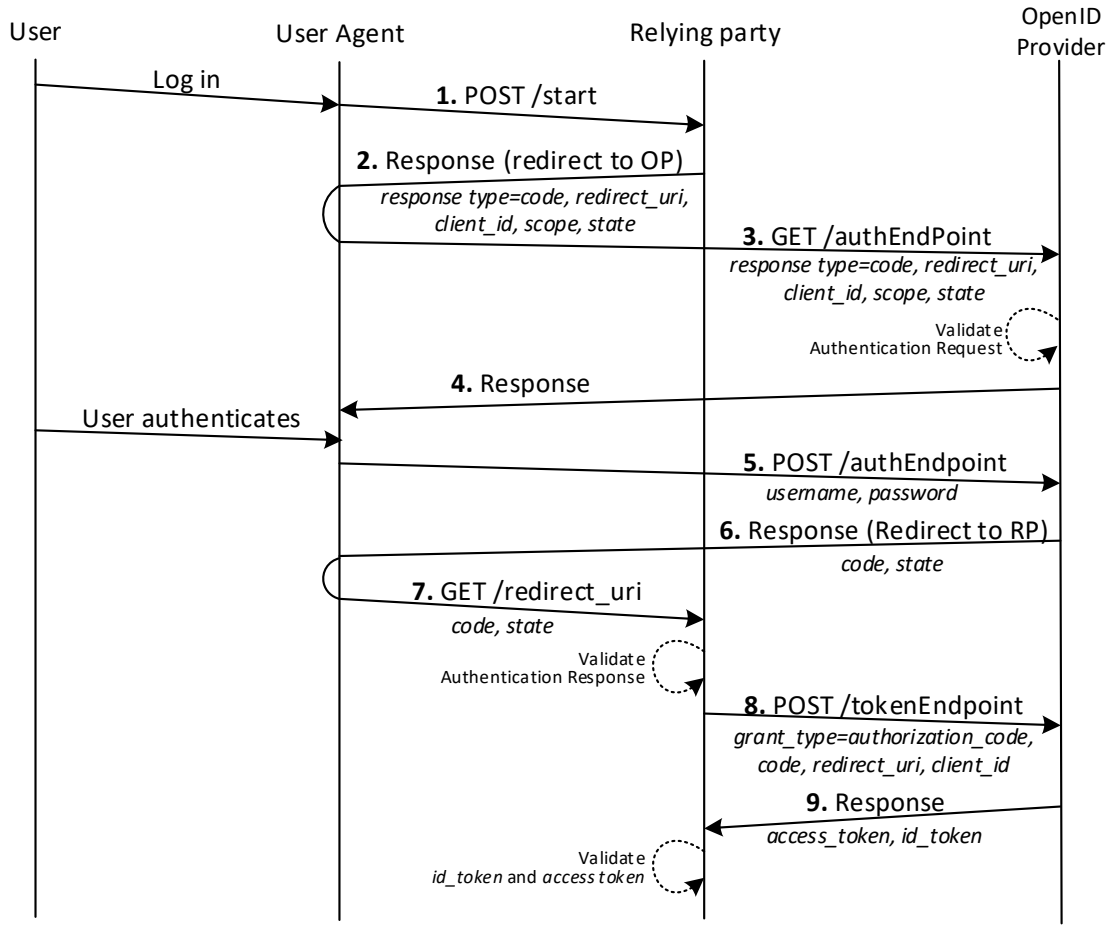

Figure 3.9: OpenID Connect Authorization Code Flow.

Step 1. The flow begins when the User clicks on the login button that triggers a request from the user agent (e.g. browser) to the RP (Message 1).

Step 2. The RP redirects the user agent to the OpenID Provider chosen by the user (Message 2 and Message 3). This message carries the following parameters: *response_type*

with the value '*code*'; *redirect_uri* to be used in messages 6 and 7; *client_id*, the RP identifier, *scope* with at least the value '*openid*'; and a recommended parameter *state*. Contrary to OAuth 2.0, the scope value is mandatory. In addition, the user agent requests the authentication web page (Message 3) provided by the OpenID Provider through the GET response containing the log in form (Message 4). However, the OpenID Provider must validate the request before answering (Message 4). For example, scope parameter has to be in the request and it has to contain the value 'openid'.

Step 3. The user provides his/her username and password. The user agent sends this information to the authorization endpoint (Message 5).

Step 4. If the user's credentials are correct and the User grants the RP's access request, the OpenID Provider sends an authorization *code* and the *state* to the user agent (Message 6). The latter is only present if it was sent in Message 3. This response message (Message 6) redirects the user agent back to the RP using the redirection URI with the parameters *code* and *state* (Message 7). These parameters are sent as query parameters to the *redirect_uri*. RP validates the message received. If the parameter state was sent in Message 3, it should also be present in this message, with the same value. Parameter code has to be present as well. All unrecognised parameters have to be ignored.

Step 5. The RP requests an id token and an access token by sending a message to the OpenID Provider token endpoint with parameters: *grant_type* equals to '*authorization_code*'; authorization *code*, received from OpenID Provider; *redirect_uri*; and the *client_id* (Message 8).

Step 6. The OpenID Provider checks the validity of the values sent by the RP. If all values are valid, then the OpenID Provider responds with the *id_token* and *access_token* (Message 9).

Step 7. Upon receiving the Message 9, the RP validates the *id_token* and *access_token*. If they are valid, then the RP can request the User's protected resources.

Chapter 4

Identity and Access Management

In this Chapter, the key concepts of the Identity and Access Management discipline are discussed. Identity and Access Management (IAM) is the process of managing who has access to what information. It is a cross-functional activity including the creation of distinct identities for individuals and systems, as well as the association of system and application-level accounts to these identities. In addition, IAM is a complex process consisting of various policies, procedures, activities, and technologies that require the coordination of many groups inside an enterprise, such as human resources and IT [17], [80]. Essentially, IAM strives to address three important issues [80]:

- Who has access to what information? IAM enables not only the management of digital identities, but also the management of the access to resources, applications, and information that these identities require.
- Is the access appropriate for the job being performed? IAM allows to control if the access is correct and defined appropriately in order to support a specific job function. Besides, IAM allows to control if the access to a particular resource conflicts with other access rights.
- Is the access and activity monitored, logged, and reported appropriately? Apart from benefiting the user through efficiency gains, IAM processes should be designed in a way supporting regulatory compliance.

4.1 Key Concepts

The key concepts of IAM are the following [80]:

- **Identity**: is the element or combination of elements used to uniquely describe a person or machine. It can be what you know (e.g., password); what you have (e.g., an ID card, security token); who you are (e.g., fingerprint pattern); or any combination of these elements.

- **Access**: is the information about the rights that the identity was granted. The access rights can be granted to allow users to perform transactional function at various levels.
- **Entitlements**: are the collection of access rights to perform transactional function.

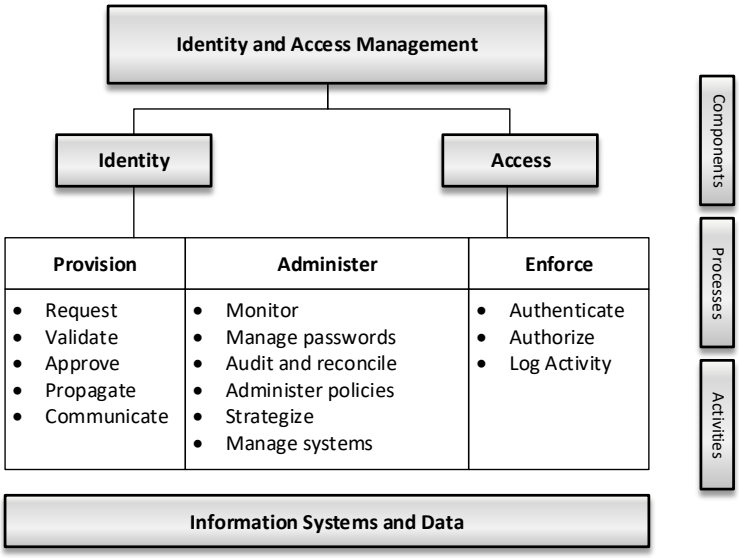

Figure 4.1: Relationship between IAM Components and Key Concepts. (Source: [80])

At this point, it is worthwhile to highlight that identities are not only related to human users. They can be also related to service accounts, machines, and other non-human entities that must be managed. For identities to become part of the organization and the access management system, they need to pass through the following stages [80]:

- **Provisioning**: it refers to an identity's creation, change, termination, validation, approval, propagation, and communication. Besides, provisioning should be governed by a company-specific and universally applied policy statement that is written and maintained by the IT department of the organization with input from other business units.
- **Identity Management**: it includes the following:
  a. the establishment of an IAM strategy,
  b. administration of IAM policy statement changes,
  c. establishment of identity and password parameters,
  d. management of manual or automated IAM systems and processes, and

e. periodic monitoring, auditing, reconciliation, and reporting of IAM systems.

- **Enforcement**: it includes the authentication, authorization and logging of identities as they are used within the organization's IT system. The enforcement of access rights mainly occurs through automated processes or mechanisms.

4.2 Identity Federation

Identity federation is an extension of identity management to multiple security domains. These domains can be internal business units, external business units, and other third-party applications and services [91]. The main objective is to provide the sharing of digital identities so that a user will be able to get authenticated one time and then access applications, services or resources across multiple domains. However, the cooperating organizations should form a federation based on agreed standards and mutual levels of trust so that they can securely share digital identities.

Federated identity management refers to the agreements, standards, and technologies that allow the portability of identities, identity attributes, and entitlements across multiple organizations and applications as well as support numerous users. In particular, when multiple organizations implement interoperable federated identity schemes, an employee in one organization can use a single sign-on to access services across the federation with trust relationships associated with the identity [91].

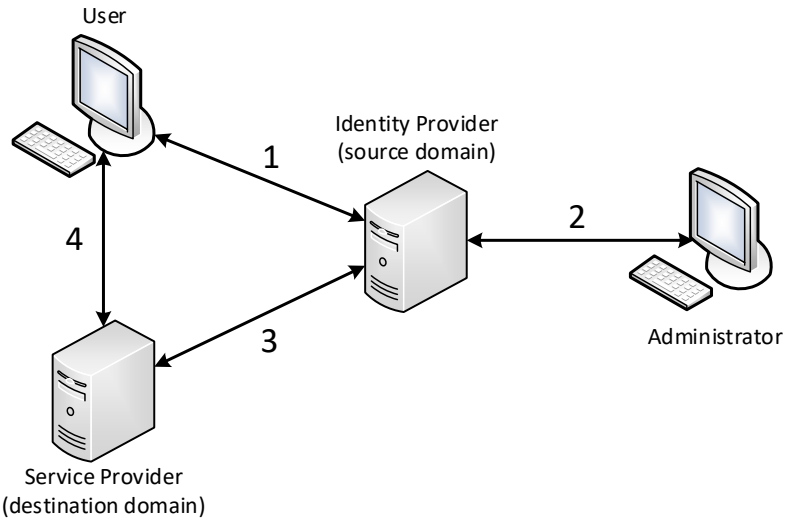

Figure 4.2: Federated Identity Operation. (Source: [91])

Besides, federated identity management is a standardized means of representing attributes. Digital identities include attributes other than simply an identifier and authentication information. For instance, attributes may include account numbers, organizational

roles, file ownership, etc. Moreover, an additional key function of federated identity management is identity mapping. Thus, the federated identity management protocols map identities and attributes of a user in one domain to the requirements of another domain [91]. Figure 4.2 shows a generic federated identity management architecture.

The Identity Provider obtains attribute information from the users (Step 1) and administrators (Step 2). On the other hand, the Service Provider obtains and employs data maintained and provided by the Identity Provider (Step 3) in order to support authorization decisions (Step 4) and to gather audit information. The Service Provider can be in the same domain as the user and the Identity Provider. However, the power of this approach is for federated identity management, in which the Service Provider is in a different domain.

Chapter 5

Implementation

This Chapter contains the implementation of three scenarios in the context of the semiconductor supply chain environment so that we can get a better understanding of: a) how TLS works, b) the key concepts of the identity and access management discipline, and c) how OpenID Connect (OIDC) works. OIDC runs on top of TLS and thus, TLS is the underlying layer of OIDC. Therefore, we study the TLS scenarios before studying an OIDC scenario.

5.1 Security Virtual Lab for TLS-based Communication in Semiconductor Supply Chain Scenarios

We provide the description of two scenarios, supporting TLS-based communication, in the context of the semiconductor supply chain environment. We captured the exchanged messages with Wireshark, a network sniffing software, and then examined them to get a good understanding of how TLS works. Section 5.1.1 describes our first scenario (i.e., Scenario 1) including TLS-based communication between Client and a Server. Furthermore, Section 5.1.2 we present our second scenario (i.e., Scenario 2) that extends Scenario 1 by adding a Server-to-Server TLS-based communication. Besides, in Section 5.1.3 describes the technical information about the scenarios. In particular, we provide description of the software we used and the configuration design decisions we took. Finally, it is worthwhile to mention that a step-by-step guide on how to setup our Virtual Machines is provided in Appendix A.

5.1.1 Scenario 1

The objective of Scenario 1 is to simulate the communication between two partners in the Supply Chain over TLS. In particular, this scenario considers the TLS-based communication between a PC (i.e., Client) of an employee of Partner A and a Server located at the premises of Partner B. Figure 5.1 shows the architecture of Scenario 1.

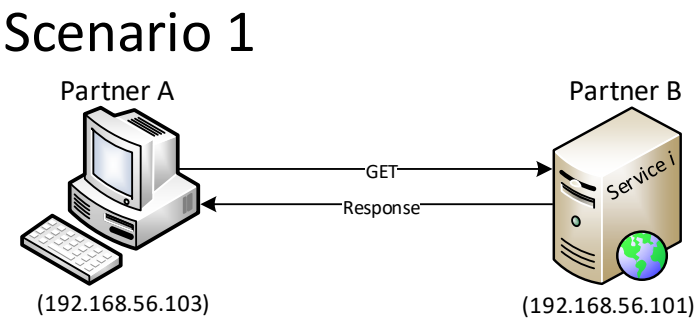

Figure 5.1: Scenario 1 Configuration.

Server of Partner B provides a Service i for the Client (i.e., user) of Partner A. This service returns a web page to the user (i.e., employee of Partner A) over TLS v1.2 protocol. However, nothing can prevent a user from typing the URL without "https://". To mitigate that, if the Client sends an HTTP request for the web page, the Server changes it to HTTPS and the communication proceeds. Figure 5.2 shows this behaviour. The communication starts as HTTP (Messages 6 and 8) but continues as HTTPS (Message 13).

No.	Time	Source	Destination	Protocol	Length	Info
2	2.177371416	192.168.56.103	192.168.56.101	TCP	74	34170 → 8080 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=1339507964 TSecr=0
3	2.178618970	PcsCompu_98:4…	PcsCompu_8f:88:…	ARP	42	192.168.56.103 is at 08:00:27:98:4a:f8
4	2.179132189	192.168.56.101	192.168.56.103	TCP	74	8080 → 34170 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=29713093
5	2.179218145	192.168.56.103	192.168.56.101	TCP	66	34170 → 8080 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=1339507964 TSecr=2971309342
6	2.180319752	192.168.56.103	192.168.56.101	HTTP	402	GET /webapp1/home HTTP/1.1
7	2.182011665	192.168.56.101	192.168.56.103	TCP	66	8080 → 34170 [ACK] Seq=1 Ack=337 Win=30080 Len=0 TSval=2971309343 TSecr=1339507965
8	2.199112266	192.168.56.101	192.168.56.103	HTTP	284	HTTP/1.1 302 Found
9	2.199153190	192.168.56.103	192.168.56.101	TCP	66	34170 → 8080 [ACK] Seq=337 Ack=219 Win=30336 Len=0 TSval=1339507969 TSecr=2971309347
10	2.369980676	192.168.56.103	192.168.56.101	TCP	74	52718 → 8443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2861886432 TSecr=0
11	2.370509680	192.168.56.101	192.168.56.103	TCP	74	8443 → 52718 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=39266010
12	2.370558206	192.168.56.103	192.168.56.101	TCP	66	52718 → 8443 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=2861886432 TSecr=3926601069
13	2.371987424	192.168.56.103	192.168.56.101	TLSv1.2	254	Client Hello
14	2.372435531	192.168.56.101	192.168.56.103	TCP	66	8443 → 52718 [ACK] Seq=1 Ack=189 Win=30080 Len=0 TSval=3926601069 TSecr=2861886433
15	3.000889381	192.168.56.101	192.168.56.103	TLSv1.2	1364	Server Hello, Certificate, Server Key Exchange, Server Hello Done
16	3.000946738	192.168.56.103	192.168.56.101	TCP	66	52718 → 8443 [ACK] Seq=189 Ack=1299 Win=32128 Len=0 TSval=2861886590 TSecr=3926601225
17	3.219184444	192.168.56.103	192.168.56.101	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Hello Request, Hello Request
18	3.219595051	192.168.56.101	192.168.56.103	TCP	66	8443 → 52718 [ACK] Seq=1299 Ack=315 Win=30080 Len=0 TSval=3926601281 TSecr=2861886644
19	3.324434159	192.168.56.101	192.168.56.103	TLSv1.2	72	Change Cipher Spec
20	3.324477000	192.168.56.103	192.168.56.101	TCP	66	52718 → 8443 [ACK] Seq=315 Ack=1305 Win=32128 Len=0 TSval=2861886671 TSecr=3926601307
21	3.325499383	192.168.56.101	192.168.56.103	TLSv1.2	111	Hello Request, Hello Request
22	3.325529663	192.168.56.103	192.168.56.101	TCP	66	52718 → 8443 [ACK] Seq=315 Ack=1350 Win=32128 Len=0 TSval=2861886671 TSecr=3926601308
23	3.387101767	192.168.56.103	192.168.56.101	TLSv1.2	435	Application Data
24	3.387850761	192.168.56.101	192.168.56.103	TCP	66	8443 → 52718 [ACK] Seq=1350 Ack=684 Win=31104 Len=0 TSval=3926601323 TSecr=2861886686
25	3.438190467	192.168.56.101	192.168.56.103	TLSv1.2	1853	Application Data
26	3.438231507	192.168.56.103	192.168.56.101	TCP	66	52718 → 8443 [ACK] Seq=684 Ack=3137 Win=35712 Len=0 TSval=2861886699 TSecr=3926601336

Figure 5.2: Monitoring the Client Request to Server's Web Page.

Figure 5.2 shows the entire communication between the Client and the Server. The next figures are excerpts of this communication. They were separated in order to highlight specific parts of the communication.

Firstly, a user enters a HTTP URL in the browser, which causes a TCP handshake to port 8080 (messages 2 and 4) and a Client acknowledge (message 5). At this point, the Client asks for the existence of the URL (message 6). The Server responds with a HTTP code 302 (message 8), meaning that the URL exists and, in the Location field, the Server informs the Client that it should use the protected URL (Please see the location attribute

from message 8 highlighted with a red rectangle in figure 5.3).

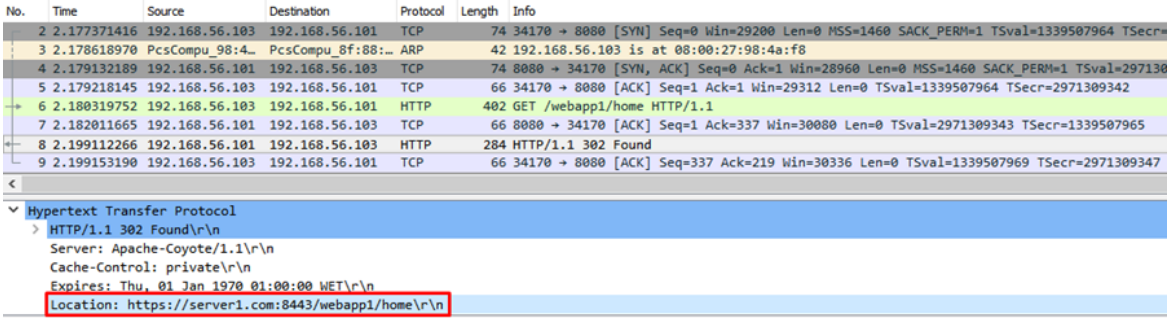

Figure 5.3: Wireshark Capture: Web Page Request from Partner A to Partner B.

In addition, figure 5.4 shows a new TCP handshake taking place to the correct secure port 8443 (messages 10, 11 and 12). From message 12 to message 22 including acknowledge messages, we have a TLS handshake to secure the transport of the Client request and the response from Server. Message 15 has a Server Hello, as well as the Server's Certificate, Server Key Exchange and Server Hello Done.

Message 23, highlighted with a red rectangle, is the request coming from the Client to the Server and, in message 25, highlighted in yellow, the Server responds with the contents of the web page shown in figure 5.5. This web page contains a button that the user in the Client machine can click to make a secure GET request to the Server over TLS.

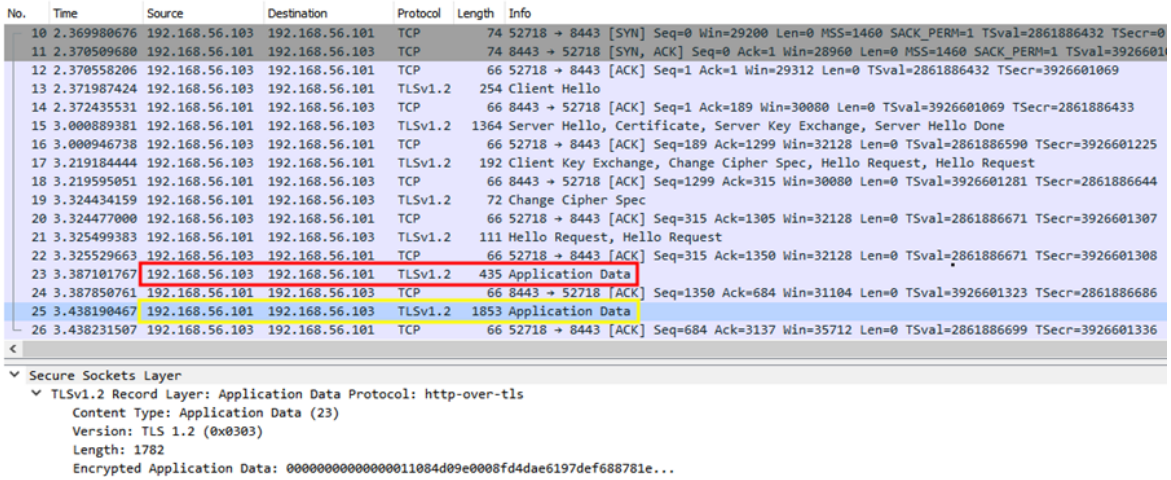

Figure 5.4: Wireshark Capture: TLS Handshake.

After the user has access to the Partner B's web page, it is possible to make a GET request over TLS. If that happens, two Application Data messages are exchanged between the Partners and two Acknowledge packets. However, in this example, the web page was loaded for a period of time before the user clicked on the button. Because of that, there is

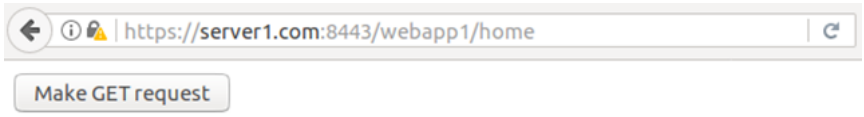

Figure 5.5: Partner B's Web Page with Button to Make GET Requests.

a new handshake. This handshake is not complete, since the user is in the same session as before. The server does not send its certificate again (message 6 is only a Server Hello).

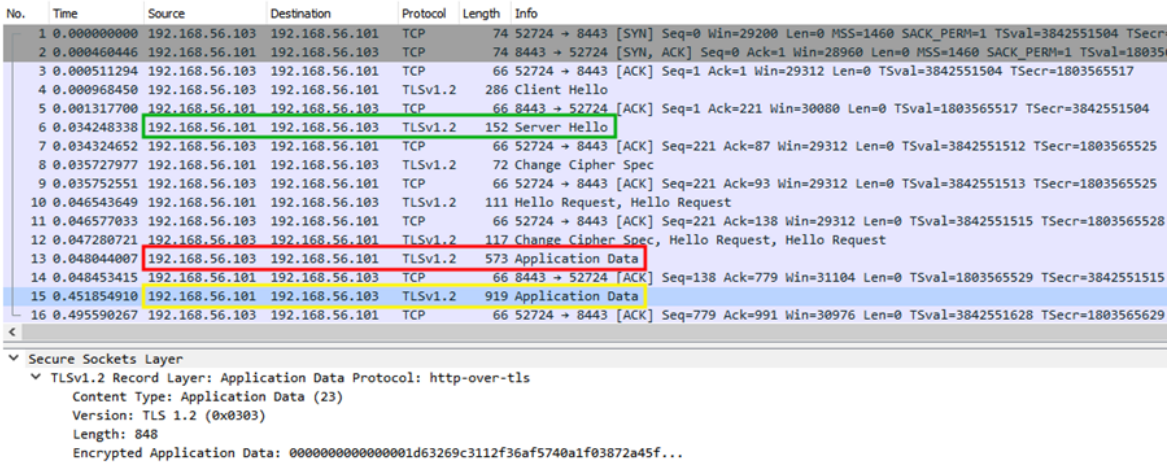

Figure 5.6: Wireshark Capture: GET Request between Partner A and Partner B.

In message 13 the GET request is sent from the Client of Partner A to the Server of Partner B. The latter responds with a new web page in message 15, as it is shown in figure 5.7.

Figure 5.7: Server Responds with a New Page.

5.1.2 Scenario 2

In this scenario, there are three Supply Chain partners (i.e., Partner A, Partner B, and Partner C). In particular, an employee of Partner A, via his/her PC (i.e., Client), gets access to Service j running on a Server of Partner C through a Service i running on a Server located at the premises of Partner B. For demonstration purposes, we have considered that the functionality of Service j is to prepend a string to the string sent by the user, through the Server of Partner B (i.e., Service i), and then to send back to the Server of Partner B (i.e., Service i). On the other hand, we have considered that the functionality of Service i is to: a) relay the request received from the user to the Server of Partner C, and b) prepend another string to the received string from the Server of Partner C and to send the result back to the user. Figure 5.8 shows the architecture of this scenario.

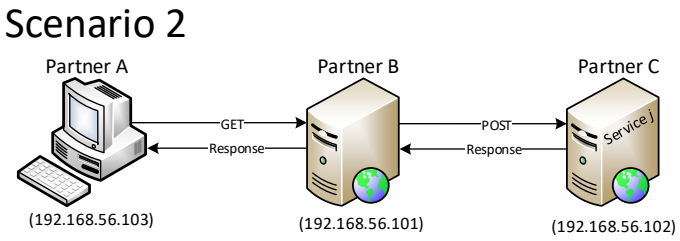

Figure 5.8: Scenario 2 Configuration.

The Server of Partner B provides a web page to the user (i.e., employee of Partner A). This web page accepts text type input ("Choose custom data") and has a button ("Access Service i"). When the user clicks on the button, a GET request is sent to the Server of Partner B. The content of the text type input box is sent along with the request.

Figure 5.9: Web Page Provided by Partner B.

The web service (i.e., Service i) running on the Server of Partner B is written in Java and is responsible to make the POST request to Service j running on the Server of Partner C. The text type input that was inserted initially by the user is transmitted along with the POST request to Service j on the Server C.

According to the defined functionality of Service i and Service j for demonstration purposes, as it is mentioned above, Service j prepends a string to the user input and sends it back to Service i running on the Server of Partner B, and then Service i prepends another string to the input from Service j. Finally, Service i sends the result back to the Client of Partner A.

The response that the user of Partner A receives is shown in Figure 5.10. In this Figure, we notice that the green rectangle includes the message sent from the Server of Partner B (i.e., Service i) to the Server of Partner C (i.e., Service j). The content of this message is the user chosen input: "Dummy message". Besides, inside the blue rectangle, we can see the message that the Server of Partner C (i.e., Service j) prepended. Finally, the red rectangle includes the string prepended by the Server of Partner B (i.e., Service i) to the output of Partner C.

Figure 5.10: Web Page from Partner B with Response from Service j.

This scenario has Server-to-Server communication over TLS v1.2 and, since we are using local servers, the Key Store of the Server of Partner C has to be added to the Trust Store of the Server of Partner B.

The Wireshark capture of this scenario starts with TLS v1.2 handshake between the Client of Partner A and the Server of Partner B, as shown in Figure 5.11, highlighted by a red rectangle. Message 6 contains the Server Hello, as well as the Server Certificate belonging to the Server of Partner B.

Figure 5.12 is the Wireshark capture after the user in Partner A clicks on the button to make a request to the Server of Partner B (i.e., Service i). Message 46 is the GET request to the Server of Partner B, containing the string from the text type input in the web page. Message 51 is the start of the TLS handshake between the Server of Partner B and the Server of Partner C. The certificate of Partner C's Server is sent in Message 53.

Messages 64 and 65, inside the blue rectangle, are the POST request from the Server of Partner B to the Server of Partner C. Afterwards, the Server of Partner C (i.e., Service j) prepends a string (for demonstration purposes) to the input received from the Server of Partner B and sends the response back to the Server of Partner B in Message 67 (yellow

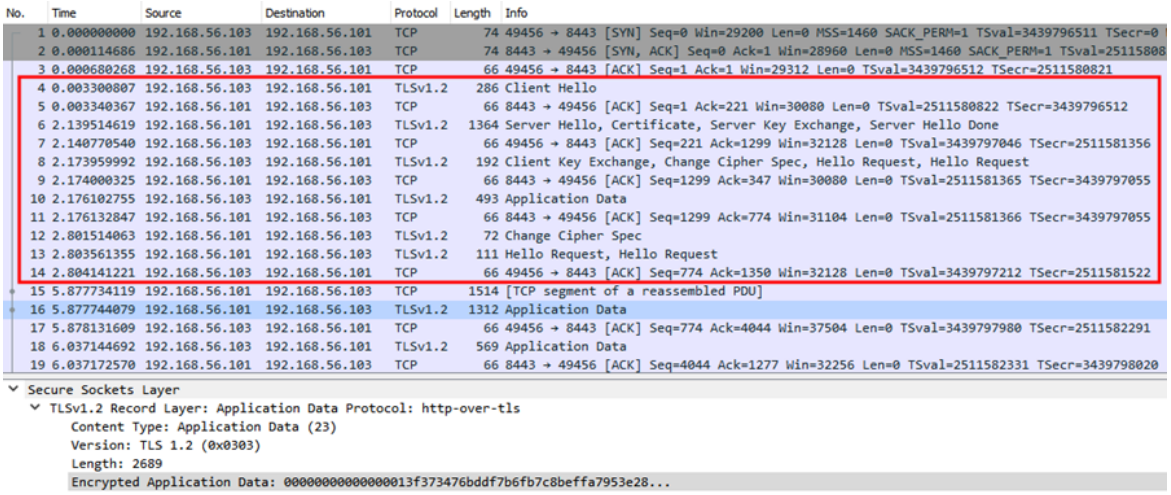

Figure 5.11: Wireshark Capture: TLS Handshake between Partner A and Partner B.

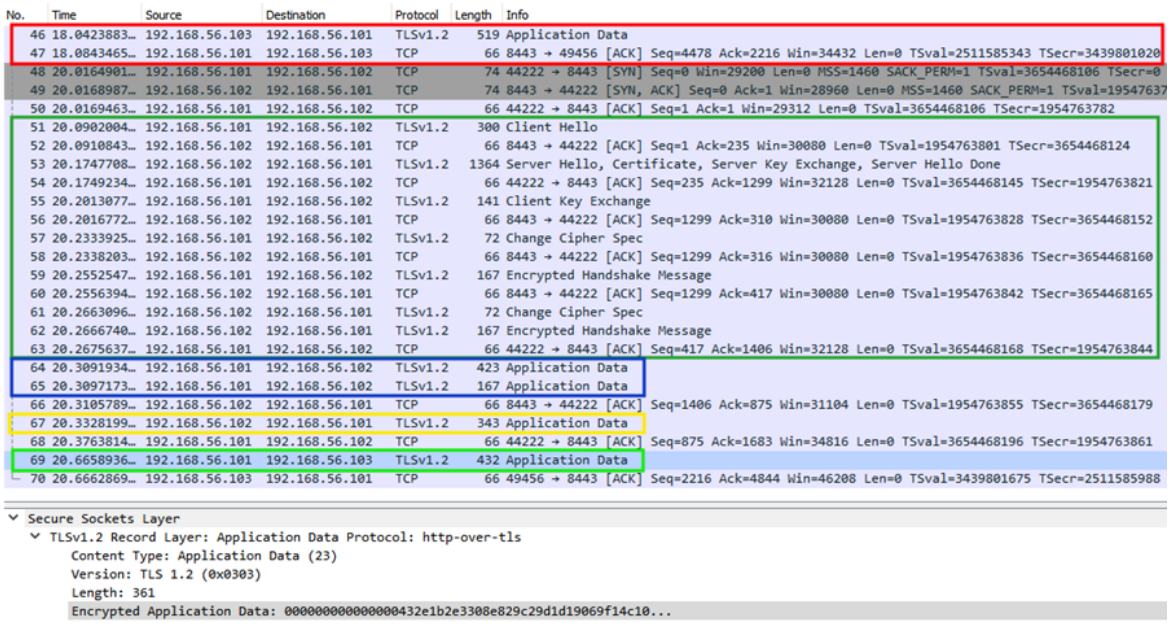

Figure 5.12: Wireshark Capture: Accessing Service j.

rectangle). Finally, message 69, inside the lime rectangle, is the response, over TLS v1.2, back to the user of the GET request he/she made via his/her PC (i.e., Client).

5.1.3 Technical Information

The two scenarios were implemented to study TLS-based communications. Scenario 1 involves a Client-Server communication and Scenario 2 extends Scenario 1 by adding a Server-to-Server TLS-based communication. To simulate each communicating party in

Scenario 1 and Scenario 2, Virtual machines (VM) were used. This is because VMs provide a cost-effective way to have multiple operating systems running on the same host machine. Specifically, we used the Oracle VM VirtualBox to create and manage guest VMs, running Linux Ubuntu 64-bit, on our host machine (Windows 10 64-bit, Intel®Core^{TM}i7-4610M CPU, 16,0 GB of RAM).

Linux was used instead of Windows, not only because it is a free and open-source operating system, but also because the size that the VMs occupied on the host machine was much smaller when they run Linux. An installation of Linux on a VM requires 10 GB of disk space, while an installation of Windows requires, at minimum, 20 GB of disk space for the 64-bit version [73].

Moreover, in both scenarios, our Servers were programmed in Java and the containers were Apache Tomcat 8 configured for TLS v1.2, as shown in Figure 5.13.

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
           maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
           clientAuth="false"
           sslProtocol="TLSv1.2"
           keystoreFile="/certs/keystoreserver2.p12"
           keystorePass="tomcatpwd"
           sessionTimeout="600">
</Connector>
```

Figure 5.13: HTTPS Configuration in Tomcat.

Each server has a Key Store containing three certificates: a) a self-signed Root certificate; b) an Intermediate certificate, signed by the Root; and c) a certificate signed by the Intermediate.

At this point, we have to mention that given the fact that the servers run on localhost, it was impossible to have a certificate signed by a trusted authority. This fact raised security issues into the Server-to-Server TLS-based communication, since a self-signed certificate could not be trusted. To address this situation, the Key Store from the Server receiving requests was added to the Trust Store of the Server sending requests. Without this step, the TLS handshake would not be performed.

Regarding the Clients, in both scenarios, they access the functionalities provided by the servers via a browser. As a browser, we used Firefox, due to the fact that it was installed by default in the VMs. Last but not least, Wireshark was used to monitor the communication between the communicating parties in both scenarios.

5.2 OpenID Connect Implementation for a Semiconductor Supply Chain Scenario

This section describes the OpenID Connect implementation for a semiconductor supply chain scenario in order to get a better understand of the key concepts of the identity and access management discipline as well as of how OpenID Connect (OIDC) works. In this scenario, an employee (i.e., user) at the Manufacturing Plant requests, through an application (i.e., Vanilla app), specific data from a service running on a Server at the Semiconductor Materials Supplier's premises. It is intended that this communication has security mechanisms that allow the sensitive information to be accessed in a secure way. Organizational aspects have also been considered, such as who is allowed to access what services or information. This is done by Keycloak, an Identity and Access Management platform, which gives roles to users and defines what that specific role can access and when. Also, Keycloak provides the implementation of OIDC in this scenario.

In section 5.2.1, the scenario is described and explained in detail. Specifically, we provide the description and explanation of two cases of the scenario: a) the case of a successful request, in section 5.2.2, and b) the case of a denied request, in section 5.2.3. Additionally, in section 5.2.4, we give the detailed description of the message flow of OIDC in the semiconductor supply chain when Keycloak is used. It is worthwhile to mention that the standard message flow of OIDC changed, as it was expected, due to the involvement of Keycloak, the Identity and Access Management platform that we considered for this scenario.

5.2.1 Scenario

Particularly, in this scenario, the user at the Manufacturing Plant connects to Vanilla, the web application provided by the Server at the Semiconductor Materials Supplier. We built Vanilla on a WildFly server by Red Hat. Assuming that it is the first time the user is accessing the application, then he/she has to get authenticated. When the user clicks on the Log In button on the Vanilla application, he/she is redirected to a Keycloak web page where he/she can get authenticated through Google. The authentication is done with OIDC and the user is redirected back to Vanilla. Then, Vanilla provides a different page to the authenticated user, where he/she can request specific data (e.g., wafer test data) by clicking on the button to access the Service running on the Server at the Semiconductor Materials Supplier.

5.2.2 Successful Request

Figure 5.14 shows the steps of the process for a successful access to the Service running on the Server at the Semiconductor Materials Supplier. It starts with the user connecting to Vanilla (Step 1). Vanilla application has two web pages, one is accessed when the user is not logged in index.jsp and the second one is for a logged in user profile.jsp.

In the index.jsp web page (Step 1), there are two buttons. One button is to log in ("LOGIN" button) and one to make a request to Service without being authenticated ("Request Service Without Permission" button). We assume the user clicked on the Login button, and then he/she is redirected to the Keycloak login page (Step 2), where the user has three options: a) to create an account, b) to log in with a Keycloak account or c) to login with a Google account. For the purpose of this scenario, the user logs in with his/her Google account by clicking the red button labelled Google at the bottom of the web page. Afterwards, the user is redirected to the Google login page (Step 3), where he/she can use his/her Google credentials to authenticate.

After authentication, the user is redirected to Vanilla application (Step 4). The web page that the user sees now has a button to log out of the application, a button to call the Service, and a box with information about the user and session (i.e., access token, id token, and principal). By clicking on the log out button the user will be redirected to the web page of Step1.

On the other hand, if the user clicks on the "Call Service" button, a request is sent to the Service running on the Server at the Semiconductor Materials Supplier with the user's access token provided by Keycloak (Step 5). For testing purposes, if the access token is valid, then the Service returns a string to the user. After receiving the string, the web page is reloaded and the message from the Service is shown: "Request is served!" (Step 6).

5.2.3 Denied Request

Figure 5.15 shows the steps of the process for an unsuccessful access to the Service running on the Server at the Semiconductor Materials Supplier. For testing purposes, we consider that the user clicks on the "Request Service Without Permission" button in the log in page (Step 1). This will send the request to the Service without an access token (Step 2) and the request will be denied.

5.2.4 OpenID Connect with IAM

Figure 5.16 shows the flow of this scenario, where OpenID Connect is used along with an Identity and Access Management platform, such as Keycloak. As illustrated in Figure 5.16, this flow consists of the following steps:

Step 1. The flow starts when the user clicks on the login button on Vanilla's initial webpage (Message 1).

Step 2. Vanilla redirects the user agent to Keycloak, where the user has to choose the OpenID Provider in order to get authenticated (Message 2).

Step 3. The user clicks on the "Google" button and triggers a request from the user agent to Keycloak (Message 3).

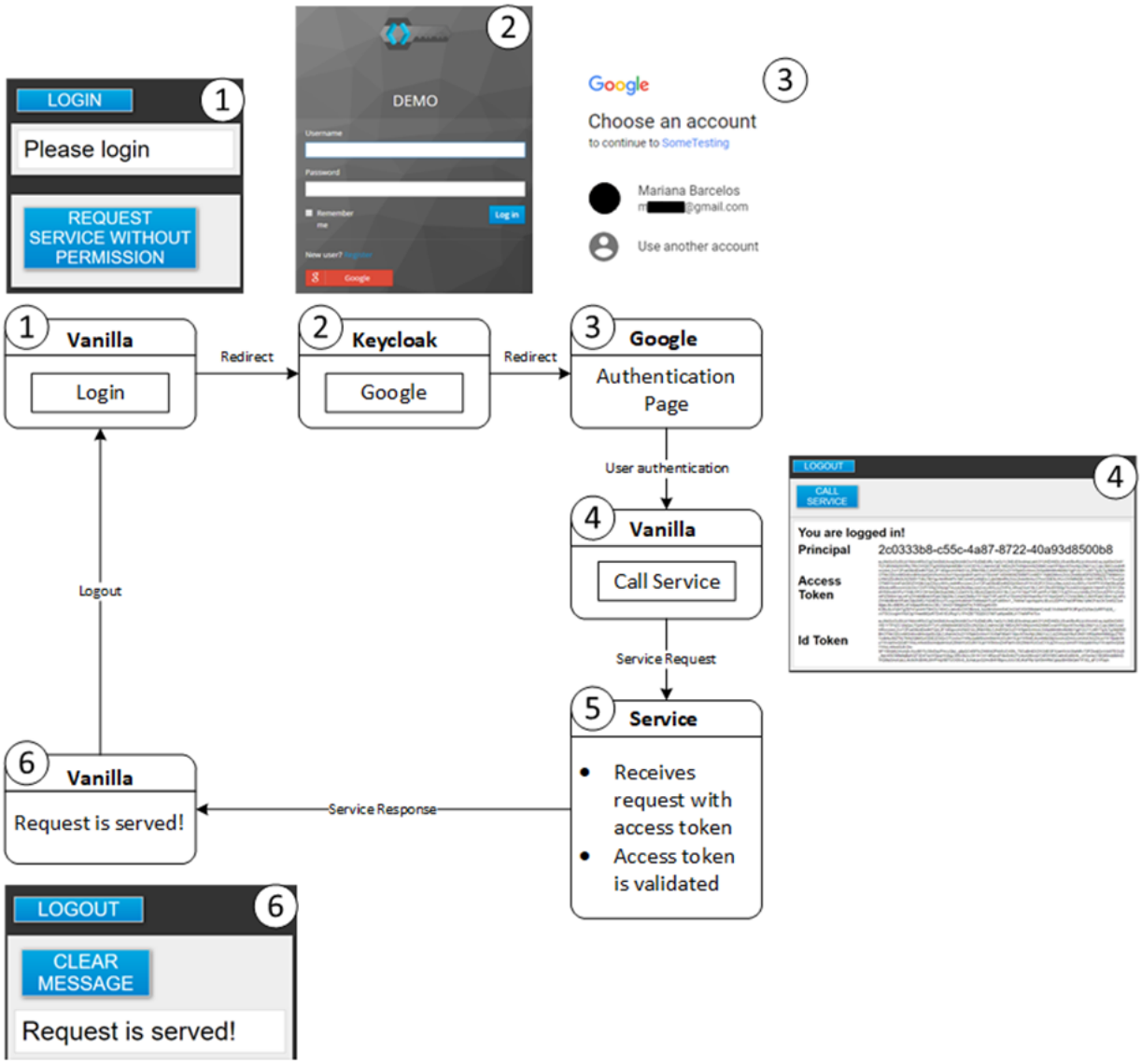

Figure 5.14: Diagram of a Successful Request to the Service Running on the Server at the Semiconductor Materials Supplier.

Step 4. Keycloak redirects the user agent to the OpenID Provider (i.e., Google) (Message 4). This message carries the following parameters: *response_type* with the value '*code*'; *redirect_uri*, *client_id*, scope with at least the value '*openid*'; and a recommended parameter *state*. In addition, the user agent requests the authentication web page (Message 4) provided by the OpenID Provider through the GET response containing the log in form (Message 5).

Step 5. The User provides his/her username and password. The user agent sends this information to the authorization endpoint (Message 6).

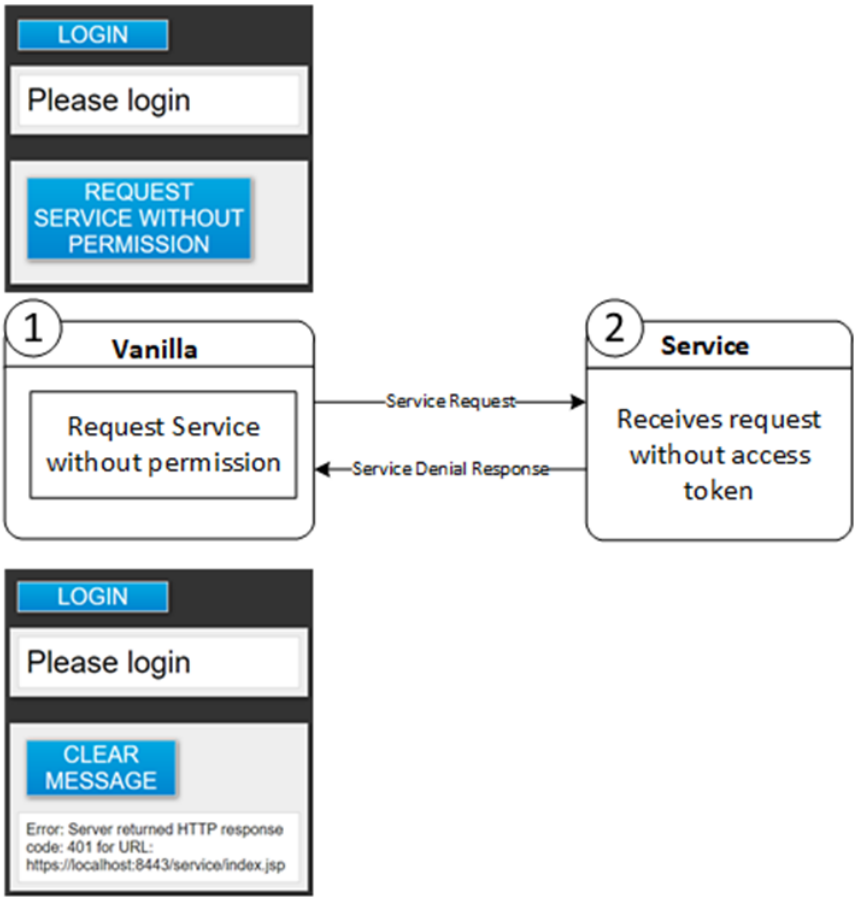

Figure 5.15: Diagram of an Attempt to Access Service without Permission.

Step 6. If the user's credentials are correct, the OpenID Provider sends an authorization code and the state to the user agent (Message 7). This response message (Message 7) redirects the user agent back to Keycloak using the redirection URI with the parameters *code* and *state* (Message 8). These parameters are sent as query parameters to the *redirect_uri*. Keycloak validates the message received. If the parameter *state* was sent in Message 4, it should also be present in this message, with the same value. The parameter *code* has to be present as well. All unrecognised parameters have to be ignored.

Step 7. Keycloak requests an id token and an access token by sending a message to the OpenID Provider token endpoint with parameters: *grant_type* equals to '*authorization_code*'; authorization *code*, received from OpenID Provider; *redirect_uri*; the *client_secret* and *client_id* (Message 9).

Step 8. The OpenID Provider checks the validity of the values sent by Keycloak. If all values are valid, then the OpenID Provider responds with the *id_token* and *access_token* (Message 10).

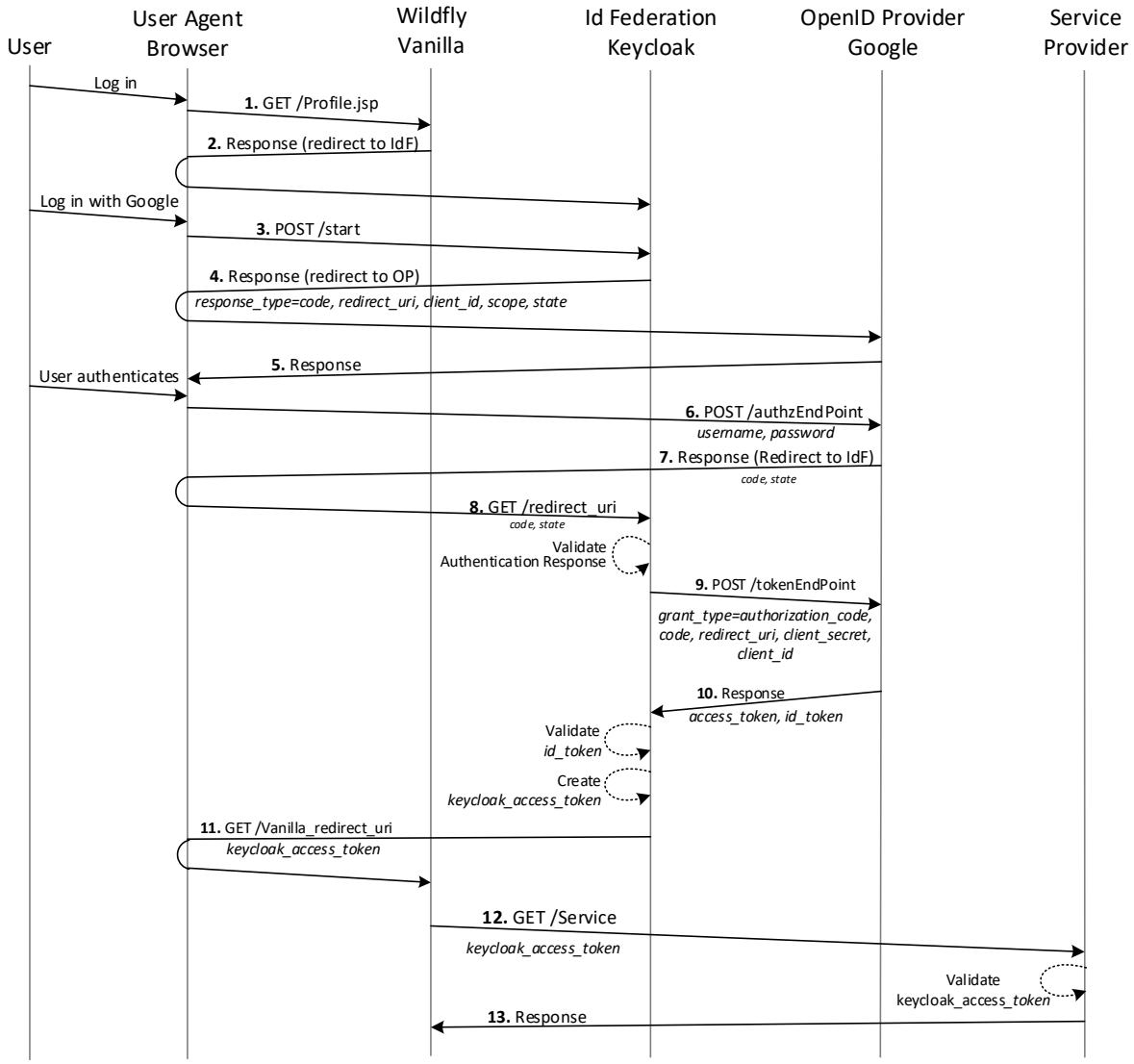

Figure 5.16: OpenID Connect Flow with Keycloak.

Step 9. Upon receiving the Message 10, Keycloak validates the *id_token* and creates its own access token (i.e., *keycloak_access_token*). Then, Keycloak redirects user agent to Vanilla (Message 11). This message carries the *keycloak_access_token*.

Step 10. Then, Vanilla can access the Service by sending to the Service Provider a request including the *keycloak_access_token* (Message 12). The Service Provider validates the token and responds back to Vanilla (Message 13).

Finally, in case that, after Step 10, Vanilla needs to access the user's protected resources stored at the OpenID Provider, the following steps, as shown in Figure 5.17, are required:

Vanilla sends the parameter *keycloak_access_token* to Keycloak to request Google's *access_token* (Message 14). In the response, Keycloak sends Google's *access_token* to Vanilla (Message 15). Afterwards, Vanilla requests the user's protected resources from the OpenID Provider's resource endpoint by sending Google's *access_token* (Message 16) and receives back the protected resources.

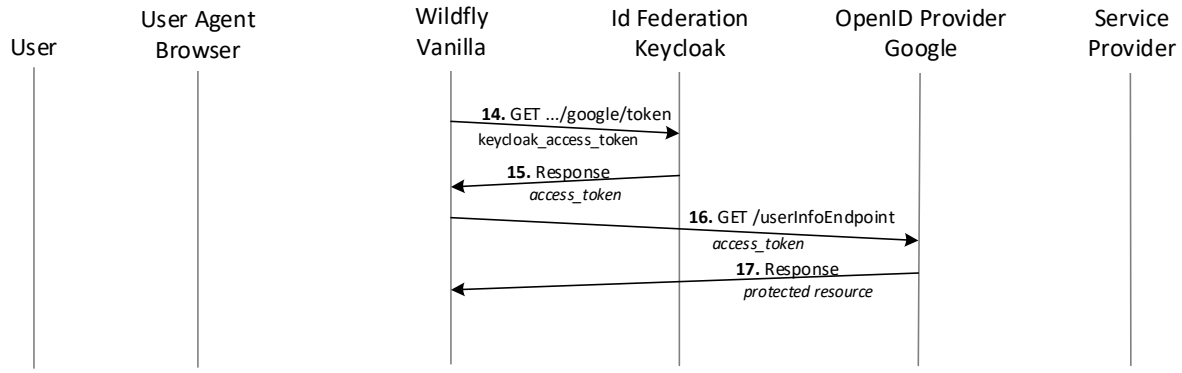

Figure 5.17: Access the Users Protected Resources Stored at the OpenID Provider.

Chapter 6

Conclusion

The work carried out during this thesis had the following main objective: To provide a foundation for a security framework for secure data communications across the partners in the semiconductor supply chain. Thus, firstly, we provided an overview of the semiconductor supply chain environment along with the description of the functionality of its main components. Furthermore, we gave a number of representative examples of various attacks that have been witnessed in the wild and can cause potential security issues and challenges in the semiconductor supply chain environment. Moreover, we studied the SSL/TLS, OAuth 2.0 and OpenID Connect and we provided their description. In addition, we provided an overview of the most well-known attacks against SSL/TLS and OAuth 2.0. The in-depth understanding of these two security protocols will allow us, as a future work towards a security framework for secure data communications, to use them as the basis for the design and implementation of more sophisticated security mechanisms that can address the specific security challenges of the semiconductor supply chain in a more efficient and effective manner. Besides, we studied and discussed the key concepts of the Identity and Access Management discipline. Finally, we implemented three scenarios related to semiconductor supply chain operations. In particular, we built a virtual security lab, using Virtual Machines (i.e., VirtualBox), where we implemented two scenarios over TLS. Afterwards, we captured the exchanged messages with the network sniffing software Wireshark and then examined them in order to get a better understanding of how SSL/TLS works. Last but not least, we implemented a third scenario on a host machine, where we used the Keycloak software, an open source identity and access management solution, in order to get a better understanding of the key concepts of the identity and access management discipline and how OpenID Connect works.

Bibliography

[1] Devdatta Akhawe, Bernhard Amann, Matthias Vallentin, and Sommer Robin. Here's my cert, so trust me, maybe? Understanding TLS errors on the web. *WWW 2013 - Proceedings of the 22nd International Conference on World Wide Web*, pages 59–69, 2013.

[2] Devdatta Akhawe and Adrienne Porter Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *USENIX security symposium*, volume 13, 2013.

[3] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. *Proceedings - IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[4] Inc. ASCO Data Security International. Diginotar reports security incident. Available at `https://www.vasco.com/about-vasco/press/2011/news_diginotar_reports_security_incident.html`. [Accessed: 9-Jan-2017].

[5] BAE Systems. Shylock Malware - a core piece of technology that is enabling wider, large scale digital criminality, 2013.

[6] BAE Systems Applied Intelligence. Shylock. Banking malware. Evolution or revolution? Technical report, BAE Systems, 2014.

[7] Tal Be'ery, Amichai Shulman, Tal Be'ery, and Amichai Shulman. A perfect crime? only time will tell. *Black Hat Europe*, 2013, 2013.

[8] Hakem Beitollahi and Geert Deconinck. Analyzing well-known countermeasures against distributed denial of service attacks. *Computer Communications*, 35(11):1312–1332, 2012.

[9] Bharat Bhargava, Rohit Ranchal, and Lotfi Ben Othmane. Secure Information Sharing in Digital Supply Chains. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pages 1636–1640, 2013.

[10] John Black and Hector Urtubia. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In *Proceedings of the 11th*

{*USENIX*} *Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 327–338, 2002.

[11] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1462:1–12, 1998.

[12] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. *Proceedings - IEEE Symposium on Security and Privacy*, pages 114–129, 2014.

[13] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[14] Junaid Ahsenali Chaudhry, Shafique Ahmad Chaudhry, and Robert G Rittenhouse. Phishing attacks and defenses. *International Journal of Security and Its Applications*, 10(1):247 – 256, 2016.

[15] Eric Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 892–903, 2014.

[16] T H Choice. THC - SSL - DoS, 2011. Available at `https://www.thc.org/thc-ssl-dos/`. [Accessed: 21-Dec-2016].

[17] Frederick Chong. Identity and Access Management, 2004.

[18] Joel Christie. Target ignored high-tech security sirens warning them of a data hack operation BEFORE cyber criminals in Russia made off with 40 million stolen credit cards, 2014.

[19] Italo Dacosta, Mustaque Ahamad, and Patrick Traynor. Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties. In *European Symposium on Research in Computer Security*, pages 199–216. Springer, 2012.

[20] George I Davida. *Chosen signature cryptanalysis of the RSA (MIT) public key cryptosystem*. Department of Electrical and Computer Science, College of Engineering and Applied Science, University of Wisc., 1982.

[21] Drew Dean and Adam Stubblefield. Using Client Puzzles to Protect TLS. In *USENIX Security Symposium*, volume 42, 2001.

[22] Antoine Delignat-Lavaud and Karthikeyan Bhargavan. Virtual Host Confusion: Weaknesses and Exploits. *Black Hat 2014 Report*, 2014, 2014.

[23] Antoine Delignat-lavaud and Karthikeyan Bhargavan. Network-based Origin Confusion Attacks against HTTPS Virtual Hosting. *Proceedings of the 24th International Conference on World Wide Web*, pages 227–237, 2015.

[24] Peter Deutsch. DEFLATE compressed data format specification version 1.3. Technical report, 1996.

[25] Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

[26] Thai Duong. BEAST. \url{https://vnhacker.blogspot.pt/2011/09/beast.html}, 2011.

[27] Thai Duong and Juliano Rizzo. Crime en la ekoparty. Available at: `https://ekoparty.blogspot.pt/2012/09/crime-en-la-ekoparty.html`. [Accessed: 22-Dec-2016].

[28] Thai Duong and Juliano Rizzo. Presentation: Crime en la ekoparty. Available at `https://www.youtube.com/watch?v=JRwPlOER6b4`. [Accessed: 22-Dec-2016].

[29] Thai Duong and Juliano Rizzo. Here Come The xor Ninjas (BEAST). *Ekoparty*, pages 1–10, 2011.

[30] Thai Duong and Juliano Rizzo. The CRIME attack. In *Presentation at ekoparty Security Conference*, 2012.

[31] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The Matter of Heartbleed. *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488, 2014.

[32] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. 4(February), 2011.

[33] Adrienne Porter Felt, Robert W Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at scale with google chrome's SSL warning. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2667–2670. ACM, 2014.

[34] Daniel Fett, Ralf Küsters, and Guido Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215, 2016.

[35] R Fielding and J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. pages 1–101, 2014.

[36] FranRosch. SSL Renegotiation Is Good. DDoS Attacks Are Bad, 2011.

[37] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, page 38, 2012.

[38] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crime attack. *Unpublished manuscript*, 2013.

[39] GReAT. The Icefog APT: A Tale of Cloak and Three Daggers, 2013. Available at `https://securelist.com/blog/research/57331/the-icefog-apt-a-tale-of-cloak-and-three-daggers/` [Accessed: 19-Oct-2016].

[40] GReAT. The Icefog APT: A tale of cloak and three daggers. Technical report, KasPersky Lab, 2013. Available at `https://kasperskycontenthub.com/wp-content/uploads/sites/43/vlpdfs/icefog.pdf` [Accessed: 19-Oct-2016].

[41] E Hammer-Lahav. The OAuth 1.0 Protocol. *Internet Engineering Task Force IETF*, 54:1–39, 2010.

[42] Dick Hardt. The OAuth 2.0 Authorization Framework [RFC 6749]. *RFC 6749*, pages 1–76, 2012.

[43] Brett Hawkings. Case Study: The Home Depot Data Breach, 2015.

[44] Paul Hoffman. RFC 3207: SMTP Service Extension for Secure SMTP over Transport Layer Security. *URL: http://ietf. org/rfc/rfc3207. txt*, 2002.

[45] Jason Hong. The State of Phishing Attacks. *Communications of the ACM*, 55(1):74 – 81, 2012.

[46] Lin-shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schechter, and Collin Jackson. Clickjacking : Attacks and Defenses. *USENIX Security Symposium*, pages 413 – 428, 2012.

[47] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. *Proceedings - IEEE Symposium on Security and Privacy*, pages 83–97, 2014.

[48] David A Huffman and Others. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[49] A Shamir I. Mantin. A practical Attack on Broadcast RC4. pages 152–164, 2002.

[50] John Kelsey. Compression and Information Leakage of Plaintext. *2365*, pages 263–276, 2002.

[51] Kim Zetter. A cyberattack has caused confirmed physical damage for the second time ever, 2017. Available at https://www.wired.com/2015/01/german-steel-mill-hack-destruction/. [Accessed: 20-Oct-2016].

[52] V Klima, O Pokorny, and Tomáš Rosa. Attacking RSA-based sessions in SSL/TLS. *Cryptographic Hardware and Embedded Systems Ches 2003, Proceedings*, 2779:426–440, 2003.

[53] Brian Krebs. Breach at Michaels Stores Extends Nationwide, 2011. Available at http://krebsonsecurity.com/2011/05/breach-at-michaels-stores-extends-nationwide/. [Accessed: 07-Nov-2016].

[54] Brian Krebs. Point-of-Sale Skimmers: Robbed at the Register, 2011. Available at http://krebsonsecurity.com/2011/05/point-of-sale-skimmers-robbed-at-the-register/. [Accessed: 07-Nov-2016].

[55] Brian Krebs. Sources: Target Investigating Data Breach, 2013. Available at https://krebsonsecurity.com/2013/12/sources-target-investigating-data-breach/. [Accessed: 03-Oct-2016].

[56] Brian Krebs. Target Hackers Broke in Via HVAC Company, 2014. Available at http://krebsonsecurity.com/2014/02/target-hackers-broke-in-via-hvac-company/. [Accessed: 13-Oct-2016].

[57] Brian Krebs. Email Attack on Vendor Set Up Breach at Target, 2017. Available at https://krebsonsecurity.com/2014/02/email-attack-on-vendor-set-up-breach-at-target. [Accessed: 17-Jul-2017].

[58] Ralph Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy*, 9(June):49–51, 2011.

[59] Ralph Langner. To Kill a Centrifuge - A Technical Analysis of What Stuxnet's Creators Tried to Achieve. (November), 2013.

[60] Robert M Lee, Michael J Assante, and Tim Conway. German Steel Mill Cyber Attack, 2014.

[61] Y H Lee, S Chung, B Lee, and K H Kang. Supply chain model for the semiconductor industry in consideration of manufacturing characteristics. *Production Planning & Control*, 17(5), 2006.

[62] Mike Lennon. Cyber Espionage Campaign Targeting Supply Chain Through Precision 'Hit and Run' Attacks, 2013.

[63] Jonathan Lewis. DDoS Attacks on SSL: Something Old, Something New, 2012. Available at https://www.arbornetworks.com/blog/asert/ddos-attacks-on-ssl-something-old-something-new/.

[64] Wanpeng Li and Chris J Mitchell. Security issues in OAuth 2.0 SSO implementations. In *International Conference on Information Security*, pages 529–541. Springer, 2014.

[65] Evariste Logota, Georgios Mantas, Jonathan Rodriguez, and Hugo Marques. Analysis of the Impact of Denial of Service Attacks on Centralized Control in Smart Cities. In *International Wireless Internet Conference*, pages 91–96. Springer, 2014.

[66] Jonathan J Lowe and Scott J Mason. Integrated Semiconductor Supply Chain Production Planning. *IEEE Transactions on Semiconductor Manufacturing*, 29(2):116–126, 2016.

[67] Subhamoy Maitra, Goutam Paul, and Sourav Sen Gupta. Attack on broadcast RC4 revisited. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6733 LNCS:199–217, 2011.

[68] Georgios Mantas, Nikos Komninos, J Rodriuez, Evariste Logota, and Hugo Marques. Security for 5G communications. 2015.

[69] Georgios Mantas, Natalia Stakhanova, Hugo Gonzalez, Hossein Hadian Jazi, and Ali A Ghorbani. Application-layer denial of service attacks: taxonomy and survey. *Int. J. Information and Computer Security*, 734(2):216–239, 2015.

[70] Moxie Marlinspike. Moxie website. Available at `https://moxie.org/software/sslstrip/` . [Accessed: 20-Dec-2016].

[71] Moxie Marlinspike. New tricks for defeating SSL in practice. *BlackHat DC, February*, 2009.

[72] Christopher Meyer and Jörg Schwenk. Lessons Learned From Previous SSL/TLS Attacks-A Brief Chronology Of Attacks And Weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.

[73] Microsoft. Windows 10 Specifications & Systems Requirements — Microsoft.

[74] Scott Mönch, Lars and Fowler, John W and Mason. *Production planning and control for semiconductor wafer fabrication facilities: modeling, analysis, and systems*, volume 52. Springer Science & Business Media, 2012.

[75] N Nelson. The Impact of Dragonfly Malware on Industrial Control Systems. Technical report, 2016.

[76] J A Orchilles. SSL/TLS Renegotiation Denial of Service, 2011.

[77] Kenneth G Paterson and Arnold Yau. Padding oracle attacks on the ISO CBC mode encryption standard. *Topics in Cryptology–CT-RSA 2004*, pages 305–323, 2004.

[78] Andrey Popov. Prohibiting RC4 cipher suites. *Computer Science*, 2355:152–164, 2015.

[79] Inc. Qualys. Trustworthy Internet ssl pulse. Available at `https://www.trustworthyinternet.org/ssl-pulse/`. [Accessed: 18-Mar-2017].

[80] S Rai, F Bresz, T Renshaw, J Rozek, and T White. Global Technology Audit Guide: Identity and Access Management. *The Institute of Internal Auditors. Retrieved from https://chapters.theiia.org/montreal/ChapterDocuments/GTAG%209%20-%20Identity%20and%20Access%20Management.pdf. Accessed*, 2, 2007.

[81] Eric Rescorla and Christopher Allen. The Transport Layer Security (TLS) Protocol — Version 1.1, 2006.

[82] N Sakimura, J Bradley, and M Jones. OpenID connect dynamic client registration 1.0 incorporating errata set 1. *URL http://openid.net/specs/openid-connect-registration-1_0.html*, 2014.

[83] Nat Sakimura, J Bradley, M Jones, B de Medeiros, and C Mortimore. OpenID Connect Core 1.0 incorporating errata set 1. *The OpenID Foundation, specification*, 2014.

[84] Pratik Guha Sarkar and Shawn Fitzgerald. Attacks on ssl a comprehensive study of beast, crime, time, breach, lucky 13 & rc4 biases. *Internet: https://www. isecpartners. com/media/106031/ssl_attacks_survey. pdf [June, 2014]*, pages 1–23, 2013.

[85] Stuart E Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 51–65. IEEE, 2007.

[86] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[87] Y. Sheffer, R. Holz, and P. Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). pages 1–27, 2015.

[88] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). Technical report, 2015.

[89] Mohamed Shehab and Fadi Mohsen. Towards enhancing the security of OAuth implementations in smart phones. *Proceedings - 2014 IEEE 3rd International Conference on Mobile Services, MS 2014*, pages 39–46, 2014.

[90] Ms. Smith. Home Depot IT: Get hacked, blame Windows, switch execs to MacBooks, 2014. Available at `http://www.csoonline.com/article/2845620/microsoft-subnet/`

home-depot-it-get-hacked-blame-windows-switch-execs-to-macbooks.html. [Accessed: 17-Oct-2016].

[91] William Stallings and Mohit P Tahiliani. *Cryptography and Network Security: Principles and Practice, 6th Edition*, volume 6. Pearson London, 2014.

[92] Victor Sucasas, Georgios Mantas, Ayman Radwan, and Jonathan Rodriguez. A lightweight Privacy-preserving OAuth2-based protocol for smart city mobile apps. *2016 IEEE Globecom Workshops, GC Wkshps 2016 - Proceedings*, 2016.

[93] Victor Sucasas, Georgios Mantas, Ayman Radwan, and Jonathan Rodriguez. An OAuth2-based protocol with strong user privacy preservation for smart city mobile e-Health apps. *2016 IEEE International Conference on Communications, ICC 2016*, (333020), 2016.

[94] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *USENIX security symposium*, pages 399–416, 2009.

[95] Dragonfly Symantec. Cyberespionage attacks against energy suppliers, version 1.21. *Mountain View, California*, 2014. Available at http://www.symantec.com/connect/blogs/dragonfly-western-energy-companies-under-sabotage-threat. [Accessed: 17-Oct-2016].

[96] Symantec Security Reponse. Security Response Dragonfly : Western Energy Companies Under Sabotage Threat. *Symantec*, 2014.

[97] Target. Target Confirms Unauthorized Access to Payment Card Data in U.S. Stores, 2017.

[98] Reha Uzsoy, Chung-Yee Lee, Martin-Vega, and Louis A. A review of production planning and scheduling models in the semiconductor industry part I: system characteristics, performance evaluation and production planning. *IIE transactions*, 24(4):47–60, 1992.

[99] Keerthi Vasan K. and Arun Raj Kumar P. Taxonomy of SSL/TLS Attacks. *International Journal of Computer Network and Information Security*, 8(2):15–24, 2016.

[100] Serge Vaudenay. Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS... *Advances in Cryptology EUROCRYPT 2002*, 2332(1):534–545, 2002.

[101] Florian Weimer and Jan Lieskovsky. CVE-2013-2191 Python-bugzilla does not verify Bugzilla server certificate, 2013.

[102] Will Gragido. Lions at the Watering Hole The VOHO Affair, 2012.

[103] RN Williams. An extremely fast Ziv-Lempel data compression algorithm. *[1991] Proceedings. Data Compression Conference*, pages 362–371, 1991.

[104] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Appendices

Appendix A

Virtual Machines Setup

This appendix shows how to setup the Virtual Machines (VMs) used in the document.

A.1 Requirements

Before the installation of the VMs, you will need some files. The VirtualBox Manager is to be installed in the host machine, but files like Apache Tomcat are going to be needed inside the VMs. You will find tips on how to use a Shared Folder between host and guest machine in Section "How to access Shared Folder".

- Oracle VM VirtualBox Manager: https://www.virtualbox.org/wiki/Downloads
- Linux Ubuntu: https://www.ubuntu.com/download/desktop
- Apache Tomcat 8: http://tomcat.apache.org/download-80.cgi

A.2 Step by Step Virtual Machines Installation

This step-by-step tutorial was made when creating a VM called Alice. On the instructions, there are placeholders for the VM name, but the pictures are related to Alice.

1. Launch Oracle VM VirtualBox Manager and select New to create a new VM.
2. Insert the VM name, type Linux and Version Ubuntu 64-bit if your host is a 64-bit version. Click Next.
3. In the memory size step it should at least 1024 MB. Select more if the host machine allows it. Keep in mind that there will be three VMs running simultaneously. Click Next.

4. Hard disk: Leave "Create a virtual hard disk now" selected and click Create.

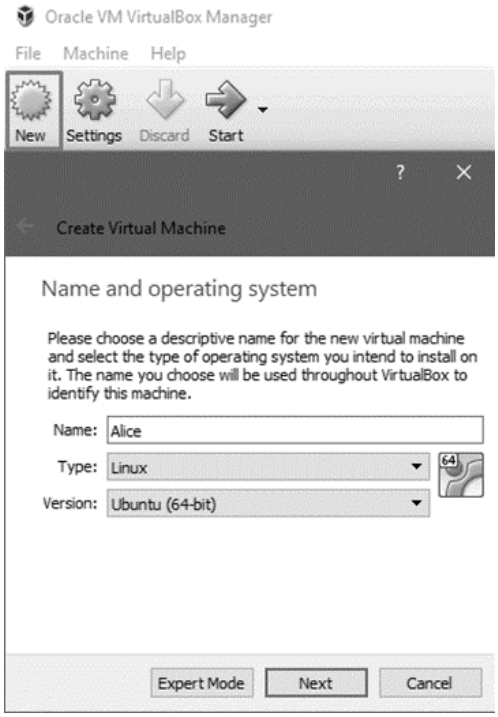

Figure A.1: Step 1 and 2: Create VM.

4.1. Leave "VDI (VirtualBox Disk Image)" selected and click Next.

4.2. Leave "Dynamically allocated" selected and click Next.

4.3. 10.00 GB are enough for this setup. Click Create.

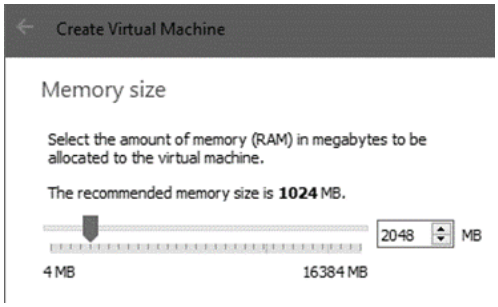

Figure A.2: Step 3: Memory Size.

5. Go to the Settings of the newly created VM Shared Folders and add a new Shared Folder. Figure A.3 has highlighted the "add" button in the top right.

5.1. Select the folder path from the host machine and the name on the guest machine (the Virtual Machine). Auto-mount should be selected. Click OK and stay on the Settings window.

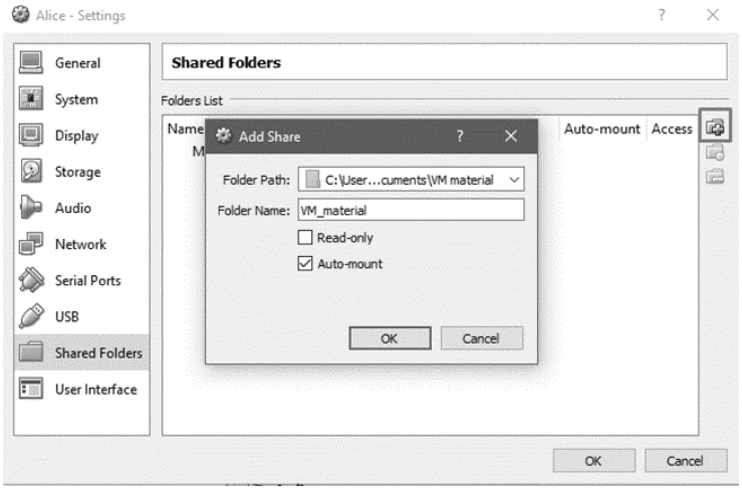

Figure A.3: Step 5: Shared Folders.

---

6. In the VM Settings, go to Network. On the first adapter leave the NAT adapter since it will be needed to install Ubuntu. Later on, we will change this and the machine will not have Internet access.

   6.1. Go to the tab Adapter 2 and check the "Enable Network Adapter" checkbox. This allows the VM to communicate with other VMs and the Host machine.

   6.2. On the attached to, select Host-only Adapter and the name should be automatically selected. Click OK.

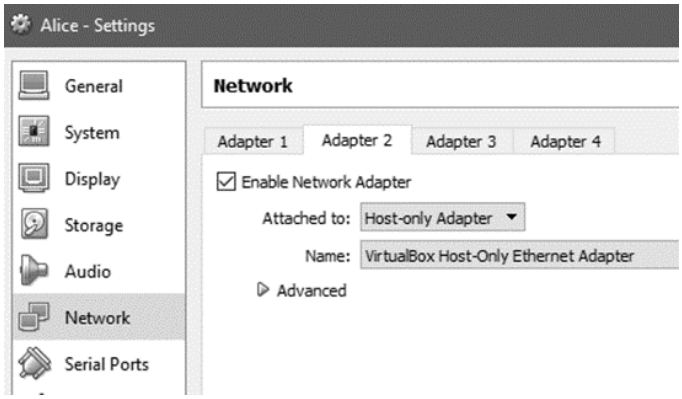

Figure A.4: Step 6: Network Adapter.

7. In the Settings window, go to Display and check the "Enable 3D Acceleration". Without this, the VM will be very slow. Click OK.

8. Start the Virtual Machine on the green arrow on the VirtualBox Manager when the VM is selected or right click the VM and select Start.

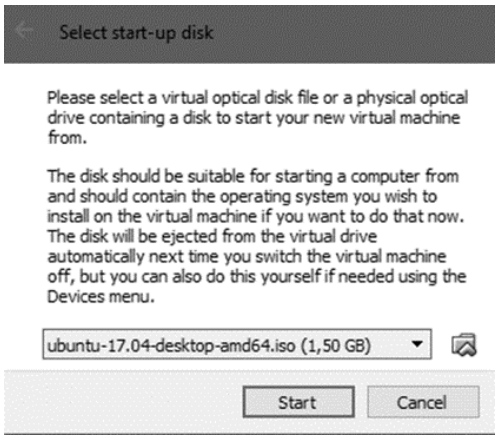

Figure A.5: Step 9: First Startup.

9. On the Select start-up disk, select the ISO file with the Ubuntu downloaded in the requirements step. Click Start and wait.

   9.1. When the window "Install" appears, select the language on the left and click Install.

   9.2. Select the checkbox "Download updates while installing Ubuntu". Click Continue.

   9.3. Leave the option "Erase disk and install Ubuntu". Click Install Now. A pop-up will appear. Click on "continue".

   9.4. Where are you? Type the VM's location and click Continue.

   9.5. Select the host machine's keyboard layout and click Continue.

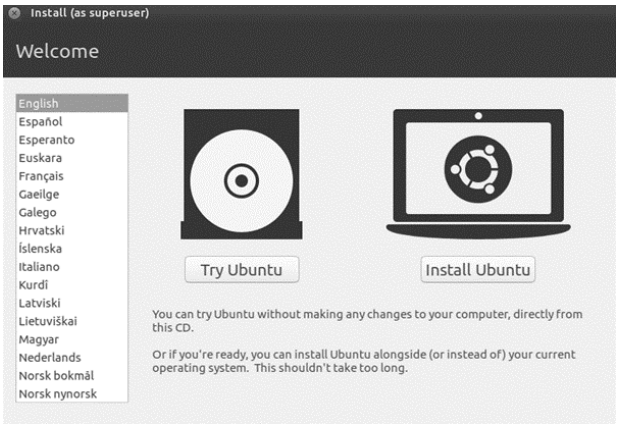

Figure A.6: Step 9: Install - Select Language.

10. Who are you? Select the computer name and password. Here are the values for Alice:

    - **Your name**: Alice
    - **Your computer's name**: alice-VirtualBox
    - **Pick a username**: alice
    - **Choose a password**: alice123

    10.1. Select your preferences in log in and click Continue.

    10.2. Wait for the window informing that the installation is complete. Click Restart Now.

    10.3. Press Enter when the screen says Please remove the installation medium, then press ENTER.

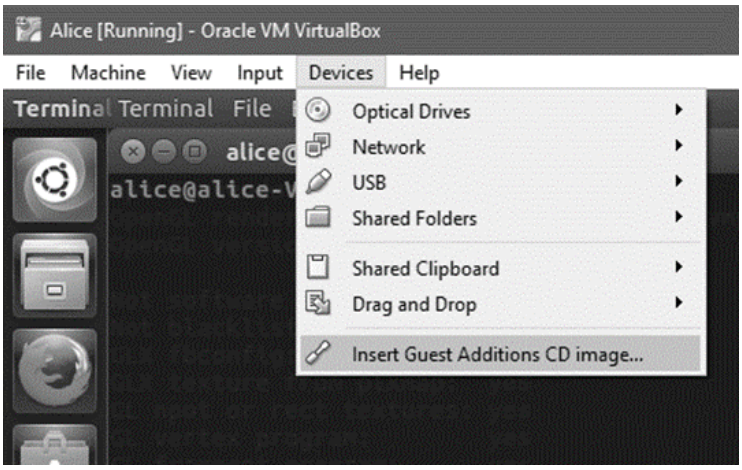

Figure A.7: Step 11: Install Guest Additions.

11. Select Devices Insert Guest Additions CD image

    11.1. Wait for the pop-up window. Click Run and type the VM's password. Click Authenticate.

    11.2. Wait for the message Press return to close this window in the Terminal and press Enter.

    11.3. Restart the VM.

12. Open the Terminal

    12.1. Type `/usr/lib/nux/unity_support_test p`

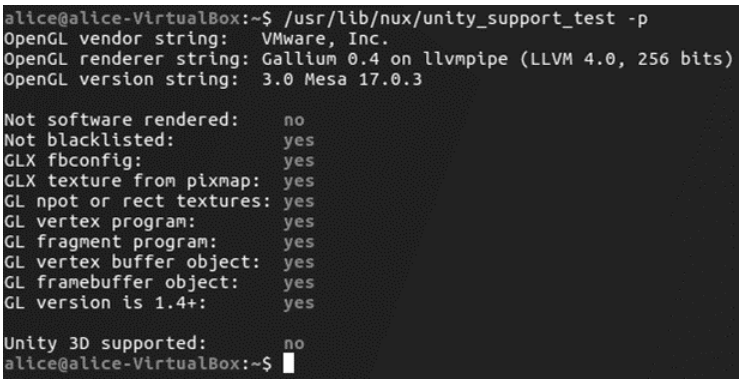

Figure A.8: Step 12: Enable 3D Acceleration.

12.2. If the list printed on the Terminal has two red "no", type `sudo apt-get install linux-headers-$(uname -r) build-essential` on the Terminal.

12.3. Then, type `sudo bash -c 'echo vboxvideo >> /etc/modules'`.

12.4. When you type again the command in step 12.1, Unity 3D supported should appear with a green 'yes'.

If the VM is a Client and only needs access to a browser, this installation is sufficient. However, you should change the Network Adapter 1 to Internal Network.

If the VM is supposed to be a server, please continue.

13. Install Java.

13.1. `sudo apt-get update`

13.2. `sudo apt-get install default-jdk`. Type 'y' without apostrophes when asked.

14. Install Tomcat.

14.1. By now, you should have downloaded the binary tar.gz from the Tomcat Download page (http://tomcat.apache.org/download-80.cgi)

14.2. Create tomcat folder with the command `mkdir /opt/tomcat`

14.3. Go to the folder where you have the tar.gz and type `sudo tar xzvf apache-tomcat-8.5.14.tar.gz C /opt/tomcat`. Keep in mind your file might have a different name.

14.4. If the unzip command created an archive folder, for example, if the bin folder is in /opt/tomcat/apache-tomcat-8.5.15/bin do the following:

14.4.1. `sudo mv /opt/tomcat/apache-tomcat-8.5.15/* /opt/tomcat`

14.4.2. `sudo rm r /opt/tomcat/apache-tomcat-8.5.15`

14.5. If `cd /opt/tomcat/bin` results in bash: cd: bin/: permission denied do:

14.5.1. `sudo chmod R 775 /opt/tomcat`

14.6. Go to /opt/tomcat/bin folder and type `sudo ./startup.sh`. Wait until full Tomcat startup (monitor log in /opt/tomcat/logs/catalina.out) and open a browser to localhost:8080 to validate that Tomcat was installed correctly.

14.7. (Optional) Install vim: sudo apt-get install vim to edit server.xml file.

14.8. Edit server.xml file to configure TLSv1.2.

14.8.1. `sudo vim /opt/tomcat/conf/server.xml`

14.8.2. In the file, find the lines you see in box 1, and substitute for the contents of box 2. Note that you have to replace the property keyStoreFile with the correct path to the key store file.

14.8.3. Place the WAR file with your webserver in the folder /opt/tomcat/webapps

14.8.4. Restart tomcat: `sudo ./shutdown.sh`, wait and then type: `sudo ./startup.sh` in the folder /opt/tomcat/bin

BOX 1:

```
<!--
  <Connector port="8443"
  protocol="org.apache.coyote.http11.Http11AprProtocol"
  maxThreads="150" SSLEnabled="true" >
    <UpgradeProtocol className="org.apache.coyote.http2.Http2Protocol" />
    <SSLHostConfig>
    <Certificate certificateKeyFile="conf/localhost-rsa-key.pem"
    certificateFile="conf/localhost-rsa-cert.pem"
    certificateChainFile="conf/localhost-rsa-chain.pem"
    type="RSA" />
    </SSLHostConfig>
  </Connector>
-->
```

BOX 2:

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11Protocol"
  maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLSv1.2"
  keystorePass="tomcatpwd"
  keystoreFile="home/alice/Documents/keystore2.p12" />
```

A.3 How to access Shared Folder

If you set up a shared folder between the host and the guest machine, you can use it to exchange files easily. The shared folder in the example is called VM_material in the host machine.

To access its files you should type `sudo su` to become super user.

Then, type `cd /media/sf_VM_material` to get inside the shared folder. Note that VM_Material is the shared folder name in Windows. Now you can copy any file you need to the guest machine (using the command `cp`). When you are finished type `exit` to quit super user mode.