Development of a Software Defined Security Perimeter

Department of Electrical and Computer Engineering

UNIVERSITY OF
THESSALY

Fotios-Dimitrios Tsokos

Supervised by:
Lalis Spyros
Antonopoulos Christos

February 2018

Περίληψη

Οι ανοιχτές πόρτες σε μία υπηρεσία μπορούν να θεωρηθούν ως αδυναμία του συστήματος καθώς επιτεθέμενοι μπορούν να αποκτήσουν πολύτιμες πληροφορίες από την εξέταση τους. Ο μηχανισμός Αυθεντικοποίησης Μοναδικού Πακέτου (ΑΜΠ) λύνει το παραπάνω πρόβλημα, εφόσων επιτρέπουν την αυθεντικοποίηση χρηστών χωρίς την απαίτηση για σύνδεση.

Η Περιφέρεια Ασφάλισης Καθορισμένης από Λειτουργικό (ΠΑΚΛ), χρησιμοποιεί το ΑΜΠ σαν μηχανισμό αυθεντικοποίησης για να δημιουργήσει ένα μηχανισμό ο οποίος ονομάζεται "Black Cloud" και παρέχει πρόσβαση σε υπηρεσίες προστατευμένες πίσω από αυτό. Ο στόχος αυτής της διπλωματικής είναι να παρουσιάσει και να υλοποιήσει τα πρωτοκολλα ΑΜΠ και ΠΑΚΛ, να σχολιάσει τη λειτουργία τους αλλά και να τα δοκιμάσει απέναντι σε κοινούς τύπους επιθέσεων.

Abstract

Open ports on a service can be a vulnerability since attackers can gain valuable information by scanning them. Allowing only authorized users to access these ports would protect a service from hackers trying to gather intellingence about it before attacking. The Single Packet Authentication (SPA) is a mechanism that solves this issue by allowing clients to authenticate themselves without using a connection. The Software Defined Perimeter is a protocol which utilizes SPA in order to provide access to protected services which are secured behind a construction reffered to as a 'Black Cloud'. The aim of this thesis is to provide implementations for the SPA and SDP protocols, prove their efficiency against port scanning and DoS attacks and present/discuss their architecture.

Acknowledgements

I am particularly grateful to Dr. Spyros Lalis for all his help, the valuable critique and the time he has spent during the development of this thesis. His guidance improved this work significantly.

I would also like to thank Dr. Christos Antonopoulos, who provided useful and much appreciated feedback. His course "Secure Information Systems" helped me understand and use most of the security mechanisms and concepts discussed throughout this thesis.

I would also like to extend my thanks to my friends Sam Courtney-Guy, who spent a vast amount of time helping me edit this thesis, Georgios-Nikolaos Sideris, for lenting me his laptop, to perform tests and measurements on, and regularly providing his feedback and Chrysanthi Stroumpouli for the countless hours she spent listening to me discuss about this thesis and for her useful ideas and suggestions.

Finally I wish to thank my family, Anastasia Tsokou, Athanasios Tsokos and Eleni Tsokou for all their support and encouragement throughout these years.

Dedication

This is dedicated to my grandma, Tasia Stamogianni, whose unconditional love always made me feel secure.

Contents

Chapter 1

Introduction

1.1 The Importance of Computer Security

Nowadays, more than ever, business and the society as a whole are linked to computer technology in multiple ways, and are therefore also strongly affected by its malfunctions and security breatches. Poor security practices may cost firms huge amounts of money [11]. They can also have a severe impact on human activities or even human lives. In order to understand the escalating demand for protection against computer attacks, one may refer to indicative recent events in the field of computing.

On May 2017 the WannaCry ransomware worm [23], a worm that either encrypts the victims data or threatens to publish them unless a ransom is paid, infected over 300,000 computers by exploiting a vulnerability in Microsoft Windows systems. This was a devastating hit to public services using outdated software and hardware. The main exploits used by this ransomware were EternalBlue and DoublePulsar, which were created by the National Security Agency and became available to the public through the Shadow Brokers group, months before the attack. Fortunately, Microsoft had released patches to fix them. Even though the WannaCry worm was dealt with, a lot of users can still be victims of the aforementioned tools. Here the key to the exploitation was the outdated software running on the victim's machines. Since a great number of users depend on obsolete operating systems, such attacks can happen again. However, this attack demonstrates a common pattern among computer threats. When a network of hosts or a host is infected, the damage is inflicted on the machine and not the user's physical self.

Unfortunately, security issues and hacking apply to many more products that are based on computer technology. Electronic security locks, for example, are widely used by hotels, public services etc. On July 2012, at the Black Hat security conference, Cody Brocious presented a mechanism able to crack an electronic keycard system developed by a leading figure in the electronic locks field [10]. What is perhaps more impressive is that the mechanism used a simple Arduino Board. In practice this means that someone with $26 (the price of an Arduino) could gain access to a victim's physical space. Along the same lines, a handful of vulnerabilities were found on software running on web cameras and baby monitors, allowing hackers to gain control of them [8]. Although only a certain brand was affected by this hack, researchers reported that other brands might have similar vulnerabilities.

Another disturbing incident recently took place at a university: several Internet of Things devices were hacked and used to create a botnet which tried to launch denial-of-host attacks on the university's network by sending multiple DNS requests [1]. Although this attack was not harmful to anything other than the university's network, the possibility of hackers gaining access to and controlling a whole team of devices can be alarming.

The aforementioned incidents were down to poorly designed and exploitable systems on the hacked products. The first example was a common network attack which aimed to acquire money from its unfortunate victims, and used sophisticated tools in order to gain access to their systems. The other examples, despite being even simpler to perpetrate, managed to get access to devices interconnected with our daily physical activities: web cameras, security locks, university appliances - in other words, Internet of Things (IoT) devices. It is crucial to develop mechanisms that prevent such incidents from happening and provide secure access to services throughout networks.

1.2 Building a Black Cloud using SDP

All of these attacks have something in common, namely the attacker knows important information about its targets such as their open ports, message types, applications running. This intelligence can be obtained by the services running on the networked machines. A solution would be for the services to restrict access only to a set of predefined clients. It is obvious, this is a static and highly impractical solution. However, it is possible that a more viable and efficient solution exists today.

In an effort to combat the risk of network attacks, the Defense Information System Agency (DISA) introduced the Software Defined Perimeter (SDP) project [6]. The project's aim is to create a so-called Black Cloud: a network that cannot be scanned by any foreign hosts while providing access to services hidden behind the cloud. A basic component of SDP is the Single Packet Authorization (SPA) mechanism [28], which is used to authorize a client before it can actually connect and access the services of a given host. More concretely, SPA allows clients to transmit authentication information and achieve authentication with another server across closed network ports. Thus, the servers can block access to all of their ports and then allow only authenticated users to connect with them.

The Black Cloud consists of a network of hosts connected together. Every host authenticates itself to the others via the SPA authentication mechanism. Each host is invisible to illegitimate hosts, since it blocks access to all of its ports, therefore the entire Black Cloud structure, since it is composed of such hosts, is theoretically invisible.

The SDP provides to its clients the ability to request access to different services from providers across the Black Cloud network. The services are hidden behind the SDP's complex structure and the clients can not access them directly but through the intermediate nodes of the system which are used as gateways.

The SDP is relatively similar to Network Access Control (NAC) [4], namely it tries to define a set of protocols/solutions in order to securely connect network nodes to each other based on certain policies. However, it can also provide support for network devices, and therefore it can be used as an IoT security mechanism. Another advantage that SDP has over NAC is that SDP can create secure communication tunnels within its structure, between applications (or services) and clients, called Dynamic Tunnel Mode. This is similar to a VPN connection and is used to combat most network attacks.

1.3 Contributions

The Waverly Labs (a tech startup located in New York City) has started providing an SDP implementation using NodeJS and Michael Rashe's *fwknop* version of the SPA Protocol. However, only some components of this project are open source and can be accessed online. Another company, founded after this thesis began, Cyxtera technologies, also provides a platform using SDP technology to provide security to their client's systems. Furthermore, Verizon has, since 2016, also supplied an SDP platform to its customers. Unfortunately, both solutions are closed source.

This thesis provides implementations of both the SPA and the SDP protocol, as lightweight, reliable and efficient open source libraries. These are written in the same programming language (Python), making them easier to debug and expand. Also, the SPA implementation is made available separately, as a standalone authorization application, or as a Python library that can be used from within other application programs and protocols. Finally,based on the SPA and SPD libraries, we provide an application that enables safe internet browsing even when using the HTTP protocol, by exploiting the dynamic tunnel mode of SDP.

1.4 Thesis Outline

The rest of the thesis is structured as follows. Chapter 2 provides a basic background in computer security, network attacks, defense mechanisms and practices. Chapter 3 discusses the SPA protocol and its implementation, while the SDP protocol (which relies an the SPA protocol) and its implementation is discussed in Chapter 4. Chapter 5 presents a simple application build by using the SDP library while it also tests the SDP implementation against the popular denial-of-service attack. Chapter 6 describes support that was developed on top of the SDP protocol, which enables HTTP-based interactions with servers that reside behind a Black Cloud. Chapter 7 provides an evaluation of the implementation, in terms of complexity and overhead. Finally, Chapter 8 concludes the thesis and discusses possible improvements and extensions.

Chapter 2

Background

2.1 Common Network Attacks

The main target of this thesis is to build a reliable security system that provides mechanisms to minimize the risk of attacks. In the following we briefly describe the commonly used network attacks.

- **Sniffing** [9]: A sniffer is a host that can monitor data sent through the network and therefore read and edit them. Sniffing Attacks can be prevented by using strong cryptography mechanisms in order to obfuscate the transmitted data(Symmetric or asymmetric cryptography such as RSA, DES, AES etc.).

- **Port Scanning** [12]: A program that scans a systems open network ports for information such as network services it provides and protocols used in order to later search for vulnerabilities. The attack is carried out by sending various packets to the targets ports (such as a simple ping message). A firewall can prevent this kind of attacks by blocking input to these ports.

- **Identity Spoofing** [14]: A host can change its address in order to pretend it is someone else. To prevent this the digital signature system, used with asymmetric cryptography, is used in order to authenticate users. Also, the Public Key Infrastructure(PKI) manages digital certificates, which are linked with different hosts, and uses them to authenticate users globally.

- **MiTm Attack** [13]: Man-In-The-Middle attack occurs when a third host intervenes between a communication and performs various actions such as disrupting or eavesdropping their exchanged packets. The third host here can pass messages from the sending party to the receiving one while using identity spoofing to change its address. Such an attack cannot always be prevented. However, encrypting the exchanged messages can minimize the inflicted damage.

- **Replay Attacks** [34]: When a MiTm attack is performed and packets are received by the sender, the next step is forwarding them to the receiver. Using this technique the attacker can acquire passwords that will be helpful in order to decrypt the encryption used. To counter this attack communication protocols include the Nonce value in their packets. A Nonce is a value that is unique for each packet. If a message is replayed then the nonce will be the same for more than one packet thus proving a malfunction in the communication system.

- **Data Modification**: The process of altering the data on a transmitted packet. Hashing algorithms (such as MD5 or SHA) are used here in order to avoid such attacks by creating the message's footprint in the form of a hash code. A received message that produces a different hash code has been altered during communication.

- **DoS attacks** [22]: Denial of services attacks are performed by sending multiple packets to host until it is unable to handle other requests. The firewall can be a helpful ally when dealing with this kind of attacks by blocking undesired input and output.

2.2 Cryptography

The need for encryption arises when two parties want to communicate with each other via an open channel and still be certain that only they can properly interpret the data and no third party can interfere in any way with the transmitted messages. In addition, it is usually desired for each side to confirm the identity of the other side. There are two main encryption techniques used, symmetric and asymmetric algorithms [32].

Symmetric algorithms depend on a shared key, which is used to encrypt and decrypt the data. Some examples of symmetric algorithms are RSA, Diffie-Helman (not for encryption or decryption but used in key exchange), El Gamal, and Elliptic Curve Cryptography [16].

In asymmetric encryption the public key is used to encrypt the data and the private to decrypt them. During communication a host shares its public key with the network and other hosts use this to encrypt the data they intend to send to it. When the data is received they are decrypted using the hosts private key. Asymmetric algorithms need more computing resources to implement compared to Symmetric ones. Due to that, these mechanisms are commonly used to exchange symmetric keys between hosts before moving on to symmetric encryption for the rest of the communication. Some of the most commonly used cryptographic algorithms are DES, AES, Serpent, Blowfish and Twofish [15].

Another requirement of secure communication is that the message transmitted is not modified in any way, be it by a malicious third party or by random channel interference/noise. Cryptography solved this problem by introducing cryptographic one-way hash functions [24]. These functions take as input data and link it with an output of obfuscated text. Given the resulting obsfucated text of an one-way hashing function, it is very difficult to obtain the text used as input to this function. In other words, hash functions provide the fingerprint of the message. A host has to send a packet, as well as its hash, while the recipient has to determine if the packet's hash matches the one that was sent alongside the packet. If not, then the packet has been modified. Some hashing algorithms are SHA, Whirpool, MD5, HMAC, and MD6.

2.3 Defense mechanisms and practices

In order to defend against the preceding attacks, computer networks use a plethora of security tools that provide safe communication between two or more parties. Most of these methods use the cryptography algorithms that were described in the previous section. Below are some common security mechanisms that provide defense against attacks in the network:

- **PKI** [18]: Public Key Infrastructure can be described as a public dictionary that links public keys with their owners, making it easy for applications to validate the origin of the sender.

- **TLS/SSL** [29]: SSL and its successor TLS are communication protocols that ensure safe transactions via the following steps: (i) initial handshake and selection of common hash and cipher functions; (ii) validation of the server's certificate on the client side (using PKI); (iii) creation and exchange of symmetric key using public key cryptography or Diffie-Helman's algorithm; (iv) communication then initiates while using the symmetric key to encrypt and decrypt data;

- **SSH** [35]: Secure SHell provides secure shell access to a remote host. It achieves that using similar practices as TLS/SSL (without the PKI step).

- **SFTP** [26]: SSH FTP (like FTP), is used for transferring files through the network. However, being built on top of SSH, it also provides reliable data streams between hosts.

- **VPN** [31]: A Virtual Private Network simulates a private network for end users even though data is sent through public or shared networks making sure that unwanted hosts can not connect or access the shared data. SSH, TLS/SSL, PKI and a variety of other protocols are used to ensure the VPN's functionality.

- **Firewall** [21]: A firewall is the software that controls incoming and outgoing network packets to a system. Through it, the system can block hazardous traffic, untrusted IP's and allow access to sensitive resources only to trustworthy hosts. The firewall consists of a number of rules (also reffered to as firewall rules), which are instructions that dictate to the system what should be done with a specific type of packet. The DROP rule, for example, is used in order to deny packets that fit a specific criteria, while the ACCEPT rule acts in the opposite way. Rules are grouped into chains. The three default chains are INPUT (manage incoming packets), FORWARD (manage packets forwarded through a system) and OUTPUT (manage packets sent).

- **Virtual Machine** [33]: A virtual machine is an OS on top of the running OS. It runs on software instead of hardware and if used as a security measure it acts as a 'decontamination chamber'.

- **Reverse-Proxy**: A proxy redirecting requests from clients to a service and vice versa. The clients do not communicate directly with the service and are unaware of its real location and protocols it may uses. Such a mechanism can protect services from attacks and help manage server load.

Some of these mechanisms (such as SSL, AES encryption, Firewall, etc) have been extensively used throughout this thesis.

Chapter 3

Single Packet Authorization

3.1 Introduction

In order to exploit a vulnerability, a hacker has to first discover and scan the target system. Open ports are a good starting point for the perpetrator since they provide a way to discover which services the target machine is providing. One may then check for and exploit corresponding service-specific security issues in a variety of ways, e.g., sniffing and/or altering transmitted data, performing man-in-the-middle attacks, denial-of-service attacks, gaining access or control the target machine via buffer overflows, etc.

The process of scanning is quite easy. The attacker can send packets to target open ports and wait for a response. Based on this response the attacker can gain valuable information about which services run on which ports. For example, *nmap* [20] is an excellent tool that automates such processes.

A service that regularly deals with client requests could eliminate the threat of scanning by using the firewall to block attackers and allowing firewall access only to legitimate clients. However in this scenario, proper users need to authenticate themselves before establishing a connection. This concept is called “authentication prior to connection”.

So far, two implementations exist that try to achieve this: Port-Knocking, and its evolution (implemented in this project), Single Packet Authorization (SPA). Both are explained in the following sections.

3.1.1 Port-Knocking

The first approach that was proposed to achieve authentication prior to connection is the so-called Port-Knocking Protocol [19]. In order to open a TLS connection with a service running on a server, the client first has to send packets to several different ports belonging to the server according to a predefined sequence: the ports to be addressed and the sequence in which they have to be addressed constitute a secret message/passphrase. The server blocks access to all its ports, but monitors the network traffic towards them. When a valid port sequence is identified, the server adds a new firewall rule to allow the sender to communicate with the machine. This process is illustrated in Figure 3.1.

However, the port-knocking sequence can be monitored and replayed by a third party (a replay-attack). Anyone over a local subnet can monitor each of the clients’ transmissions, and then replay the exact same packets to the same ports in order to gain access to the system, as illustrated in Figure 3.2.

Another problem is that the effective information passed from the client to the server is

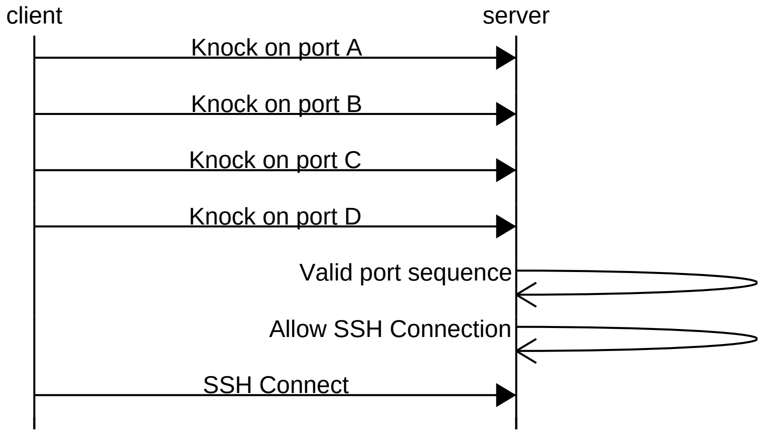

Figure 3.1: Indicative port knocking sequence.

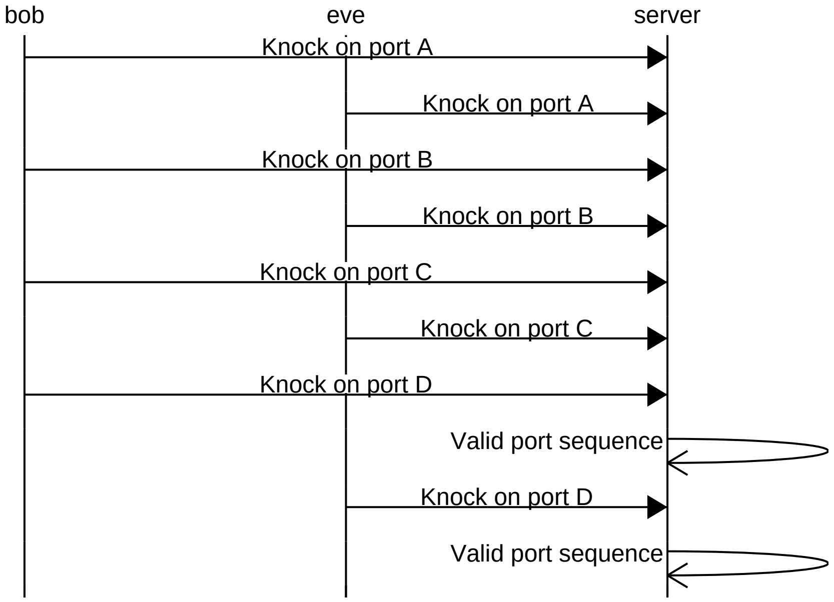

Figure 3.2: Replay attack for port knocking.

the port field of the TCP and UDP protocols, which is only 2 bytes long. This means that in order to support a secret message or passphrase of size B one must make B/2 separate packet transmissions. In fact, the process takes even longer as one needs to introduce a time delay between these transmissions, in order to avoid any out-of-order delivery of the sequence packets.

Last but not least, client authorization may fail (the server may consider the client as unauthorized) even if it provides the right sequence of knocks, if a second malicious user spoofs fake port-knock requests at the same time. This attack is not as sophisticated as a replay attack, but can cause trouble with very little effort/intelligence on the part of the attacker.

3.2 Introduction to the SPA

A new mechanism, the so-called Single Packet Authorization (SPA), which has similarities to the port-knocking concept, was presented at the Black Hat conference in 2005 by two research groups (MadHat unspecific and Simple Nomad). In the following years, Michael Rash used the SPA concept to produce his own implementation (open source, available on Github) of the protocol, called *fwknop*. *fwknop* is currently the most popular SPA implementation and a part of the work produced during this thesis has been inspired by it.

3.2.1 The SPA Protocol

Like port-knocking, SPA achieves the authentication of a client to a server. It is assumed that they both own a shared secret seed, which is used for the encryption and decryption of the exchanged messages.

The server behaves in a similar way to port-knocking. It blocks access to all ports, and then passively monitors the network for incoming packets. However, instead of port knocks, the server looks for SPA packets. The server also adds new firewall rules to allow authenticated clients to connect with it. The proccess is illustrated in Figure 3.3.

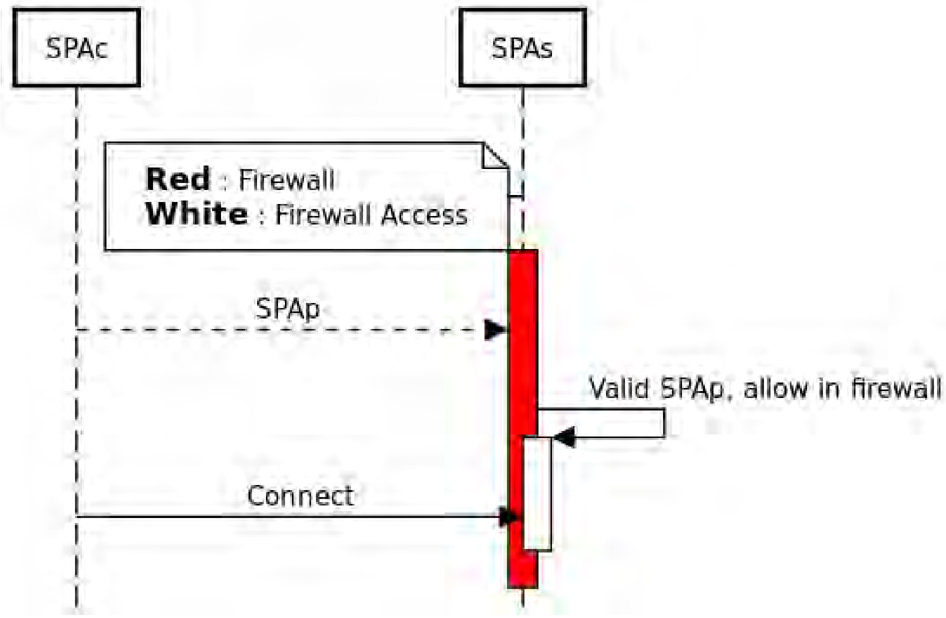

Figure 3.3: The SPA protocol

The client sends an SPA packet when it wants to access a specific port on the server. As in port-knocking, the port to be made eventually accessible to the client, is assumed to be agreed a priori by both the client and the server. More specifically, the SPA packet includes the following information:

- AID : the unique ID of the client that sends this packet
- RANDOM : a random alphanumeric produced by the client before sending the packet
- PASSWORD : the password of the client (also known by the server)
- NEW_SEED : the new value of the shared seed for the next transaction
- MD5_HASH : a hash of the previous values

These values are encrypted using the shared secret seed. The SPA packet consists of the encrypted values along with the AID of the client that is sent in plaintext (used by the server to find the right key for decrypting the packet's contents).

When the server receives a packet that has the expected SPA format, it identifies the client that sent the packet by using the AID value which is encapsulated in the packet. The server then uses the shared key that is associated with the client to decrypt the values included within the packet. If the decryption of the values was succesful, the authentication of the client completes if the PASSWORD in the packet matches the one that is known to server for that client, the hash of the received values matches the MD5_HASH value received, and the RANDOM value has not been used in a previous SPA packet sent by this client. After the authentication of the client, the server grants access to the predefined port, and sets their shared secret key to the NEW_SEED value received.

3.2.2 Changes to original Protocol

In the original SPA protocol the NEW_SEED value is not included in the SPA packet. It was introduced in this version of SPA in order to provide greater security in the SPA architecture.

SPA packets are exchanged through a public network where any attacker can sniff and capture them. Given that the AID is in plaintext, an attacker that captures the SPA packet knows the identity of the client that sent it. Then, the attacker can perform a *Known Plaintext* attack, using as input the SPA packets captured in order to obtain the client's seed and PASSWORD. This attack, although quite demanding in terms of time and processing resources, is possible, and in fact it is already widely used against other protocols.

By introducing the NEW_SEED value in the SPA packet, and by letting the server change the seed to NEW_SEED after each successful authentication, Known Plaintext attacks are deemed useless. Even if the secret seed used to encrypt an SPA packet is obtained by the attacker, the next SPA packet sent by the same client will use a different seed for encryption.

3.3 The SPA implementation

In order to test and study SPA, a Python implementation of the protocol was developed, called *spa_lib*. This section provides a detailed presentation of the structure of the implementation and the mechanisms used during the development of this library. The *spa_lib* application programming interface (API) will not be explained in detail in this section. A list and description of the classes and functions provided by this library is given in Appendix A.

3.3.1 SPA server : Listener Daemon and Firewall

The server part of SPA is a daemon thread that runs in the background. The *spa_lib* provides the *SPAListener* class, which is the representation of the server. This class extends the Python *Thread* class (as defined in Python's threading library). To start the execution of the *SPAListener* daemon thread, users of *spa_lib* should initiate a *SPAListener* object and later call its *start* function. Once started, this daemon thread will start to monitor the network for SPA packets, authenticate the clients that sent those packets, and configure its firewall accordingly. In addition, the *SPAListener* also provides additional functions that can be used to manually change the firewall.

The *SPAListener* maintains a file structure used to store client-related information such as: the cient AID, the PASSWORD, the encryption seed(s), and RANDOM values used in previous SPA packets.

The daemon thread uses the Linux *iptables* command to configure the firewall of the host machine. Initially, it adds a rule to deny/block incoming packets to *every* port of the system. Two firewall rules can be used for this purpose: the DROP and the REJECT rule. Both lead to packet being dropped, but the REJECT rule also responds with an PORT_UNREACHABLE packet to the sender of the dropped packet. Our implementation uses the DROP rule in order to have a "silent" system that is more difficult to trace by an attacker (else, the attacker would be able to infer that the machine exists and is running, and that its ports are being blocked explicitly by a firewall rule).

Everytime the daemon thread authenticates a client, an ALLOW rule is added to the firewall for the IP address of the client. This rule is configured to allow new or already established connections. Each rule in the firewall is linked with a label, so that the *SPAListener* can edit or delete existing rules by refering to it.

The *spa_lib* also provides the *add_firewall_rule* and *delete_firewall_rule* functions, which can be used to insert new rules or delete existing rules in a "manual" way. Another useful function is the *set_rule_established* function, which sets an existing rule to accept packets only from established and not from new connections.

3.3.2 The SPA packet

The SPA packet is a UDP packet sent through the network to a random port with destination the IP address of the server. The structure of an SPA packet is given in Table 3.1.

Table 3.1: Structure of SPA packet

Name	Size	Position	Description
AID	32B	b0-b31	The unique ID of the client
Delimiter	1B	b32	The ':' character
ENC	152B	b33-b184	The encrypted DAT string
Delimiter	1B	b185	The ':' character
MD5_HASH	16B	b186-b221	The hash of the packet

When the client creates the packet, the values to be encrypted (AID, PASSWORD, NEW_SEED, RANDOM) are packed into a single string (here, referred to as DAT), which is then encrypted using the SHA-256 symmetric key algorithm using the secret seed which is shared with the server. The output of the encryption consistutes the ENC field of the SPA packet. The MD5_HASH is produced by using the DAT string as input to the MD5 hashing function.

3.3.3 Sending SPA packets

The *spa_lib* provides the *send_spa* function in order to send SPA packets, which is described in pseudocode below:

Algorithm 1: send_spa function

```
Data: aid, password, seed, new_seed, server_ip
random = create_random();
dat = pack_into_string([aid, password, new_seed, random]);
enc = aes_encrypt(dat, seed);
md5_h = md5(dat);
to_send = pack_into_string([aid,':',enc,':',md5_h]);
rand_port = get_random_port();
UPD.send(ip, rand_port, data = to_send);
```

3.3.4 Checking if client is authenticated

The *spa_lib* also provides the *port_is_open* function, which checks if a certain port on the server is open and ready to accept connections. This function must be called after the execution of the *send_spa function* to check whether the SPA authentication has been successful. In case the client was indeed authenticated, the user of *spa_lib* is expected to change the secret seed they share with the server to the NEW_SEED value. An example of using the *spa_lib* to send a SPA packet is presented below:

Algorithm 2: Use of the spa library

```
Data: aid, password, seed, new_seed, server_ip, server_port
new_seed = produce_new_seed(size=32);
nof_tries = 0;
while 1 do
    spa_lib.send_spa(aid, password, seed, new_seed, server_ip);
    /* in case the server has allowed access to the firewall exit the loop
       */
    if spa_lib.port_is_open(server_ip, server_port) then
        break;
    end
    /* loop until maximum number of failed retries                    */
    if nof_tries > MAX_TRIES then
        /* unreachable server                                          */
        exit();
    end
    nof_tries += 1;
end
/* change user info, set seed for next communication to new_seed      */
edit_my_info(seed=new_seed);
```

3.3.5 SPA server : SPA packet authentication

The *SPAListener* daemon thread handles SPA authentication. When it captures a packet from the network, it checks to see if it has a structure that matches the expected SPA packet structure (as defined in the previous subsection). If this is not the case, the packet is rejected immediately. Otherwise, the *SPAListener* starts the authentication proccess. As a first step, it uses the seed it shares with the client that sent the packet, to decrypt the encrypted string ENC. This produces the DAT string, which contains the AID, PASSWORD, RANDOM and NEW_SEED values. Afterwards, the *SPAListener* checks if the PASSWORD value received matches the password of the corresponding client (AID). Then, it computes an MD5 hash of the DAT string, and checks if this is equal to the MD5_HASH that was included in the SPA packet. By comparing the two MD5 hashes, the *SPAListener* checks if the encrypted information was not modified by a third party during the travel of the packet.

The *SPAListener* also keeps, for each client, a list of RANDOM values that were included in previously sent SPA packets. In case an SPA packet from an client contains a RANDOM value that already exists in this list, the packet is considered as a replay packet and authentication fails. If the credentials of the SPA packet are valid and the client is authenticated, the *SPAListener* adds a rule to its firewall to accept traffic from the packets' originator IP address (acquired from the *src* field of the UDP packet), and adds the RANDOM value of the packet to the list of used RANDOM values for this clients. Figure 3.4 depicts this proccess.

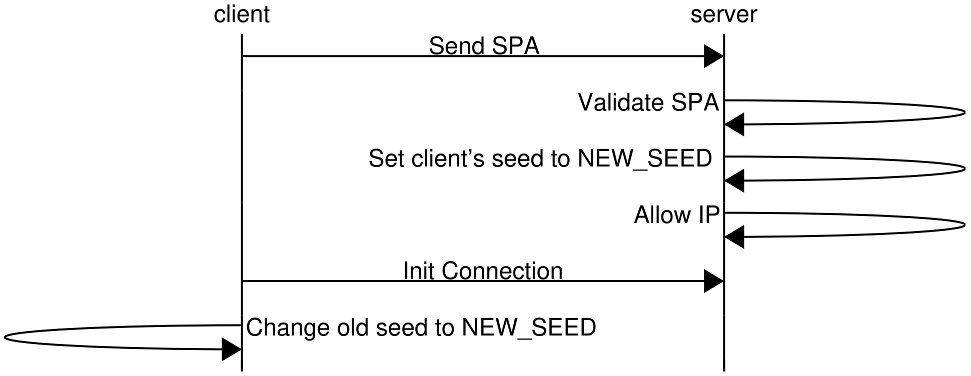

Figure 3.4: Successful SPA authentication

A problematic scenario may occur if right after the authentication succeeds, and the *SPAListener* changes the secret seed it shares with the client to the NEW_SEED value included in the last SPA packet received, one of the two parties terminates. In this scenario, the shared secret key will differ between the server and the client and the authentications attempts by the client will fail (as displayed on Figure 3.5). To deal with this problem, the *SPAListener* always stores two seeds for each client: the current shared secret seed, and the one used in a previous transaction, referred to as the OLD_SEED. If the authentication fails for the current seed, the *SPAListener* repeats the authentication process using the OLD_SEED to decrypt the packet.

The overall authentication process is presented in the algorithm below.

Algorithm 3: SPAListener Daemon Thread execution

```
Data: client_files, firewall
while 1 do
    packet = sniff_network();
    if !is_spa_format(packet) then
        continue;
    end
    spa = parse_spa(packet);
    client = client_files.get_user(spa.get_aid());
    seed = client.get_seed();
    pwd = client.get_pwd();
    dat = spa.decrypt(seed);
    e_aid, d_pwd, random, new_seed = unpack_from_dat(dat);
    /* authenticate client                                  */
    if e_aid != aid OR d_pwd != pwd then
        /* if authentication fails repeat with old seed     */
        dat = spa.decrypt(seed);
        e_aid, d_pwd, random, new_seed = unpack_from_dat(dat);
        if e_aid != aid OR d_pwd != pwd then
            continue;
        end
    end
    md5_h = md5(dat);
    e_md5 = spa.get_md5();
    /* check for modification                               */
    if md5_h != e_md5 then
        continue;
    end
    /* check for replay                                     */
    if (client.random_replay(aid, seed, random)) continue;
    /* add new seed, mark current seed as old               */
    client.add_seed(new=new_seed, old=seed);
    /* add random value to be linked with the old seed      */
    client.add_random_to_seed(seed, random);
    /* authentication complete, allow client in firewall    */
    ip = get_ip_from_packet(packet);
    firewall.allow(ip);
end
```

3.4 Discussion

The SPA protocol inserts an additional layer of security to a system, and it can be used to protect and secure multiple kinds of services. To achieve this, it makes extensive use of the firewall. The different layers of security created by the SPA are illustrated in figure 3.6.

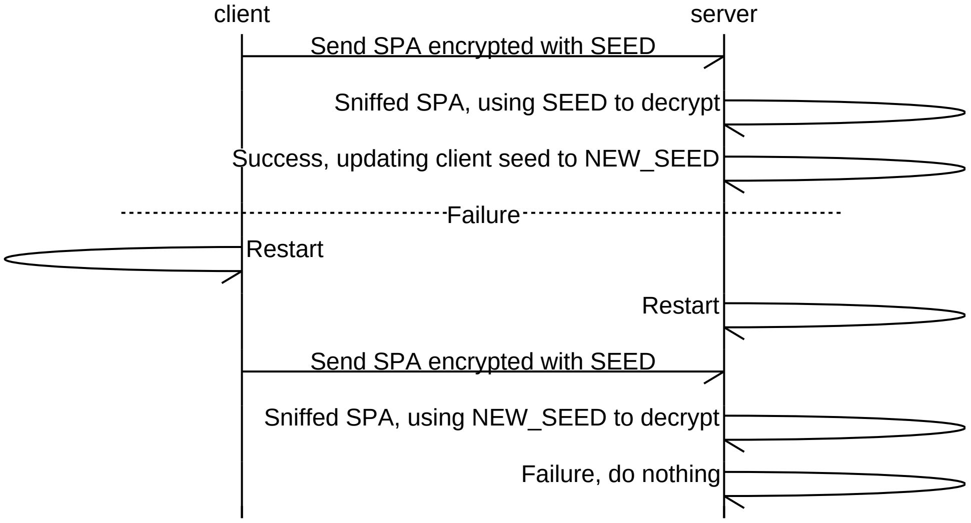
client
server
Send SPA encrypted with SEED
Sniffed SPA, using SEED to decrypt
Success, updating client seed to NEW_SEED
Failure
Restart
Restart
Send SPA encrypted with SEED
Sniffed SPA, using NEW_SEED to decrypt
Failure, do nothing

Figure 3.5: NEW_SEED change - Problematic scenario

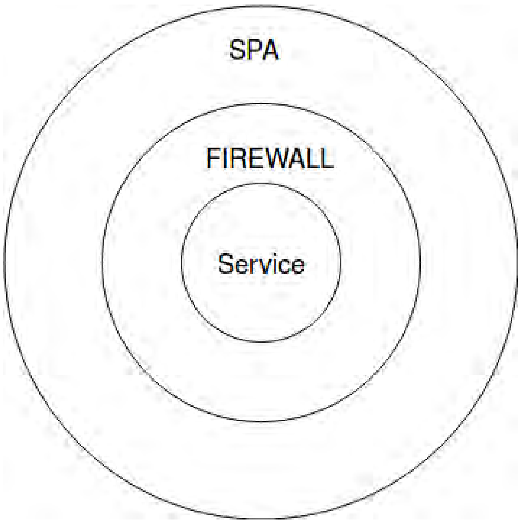

Figure 3.6: SPA : An additional security layer

3.4.1 SPA vs Port-Knocking

SPA is a solution that maintains the basic ideas of port-knocking, such as authorization prior to connection and untraceable ports, while trying to address some of its drawbacks.

Some of *SPA's* benefits vs. port-knocking are:

- Immunity against Replay-Attacks : By using the RANDOM field, SPA is impervious to Replay attacks.

- Fewer packets : Only one packet sent, compared to the number of packets required by the port-knocking sequence.

- Packet sniffing is harmless : Even if the contents of the SPA packet are acquired through sniffing, the data would be of no use to the attacker since they would be encrypted. However, sniffing port-knocking packets reveals the port-knocking sequence.

- IP spoofing can not affect authentication : By using IP spoofing, an attacker could disrupt the port-knocking authentication proccess of a client. However this does not affect the authentication of clients.

Overall SPA can be considered as an improvement to the port-knocking protocol.

3.4.2 Limitations of SPA

Even though SPA seems to provide security against most computer network attacks, high skilled hackers can still exploit the system. An attacker could monitor the network until an SPA authentication attempt is made by a client, wait until the server authenticates the client and allows it in its firewall, and then use IP spoofing to hijack the session. In that case, the problem is solved if the protected service does its own authentication check. However the attacker, instead of trying to hijack the client's service can also perform a DoS attack on the open port by using IP spoofing. So, the SPA is also susceptible to DoS attacks, but only advanced and skilled hackers can perform them. This issue is also addressed in the next chapter, as SPA can not solve it using its current structure.

Chapter 4

Software Defined Perimeter (SDP)

As explained earlier in this thesis, open network ports in a system can lead to a multitude of separate attacks. SPA was introduced in order to make authentication before connection to a server possible, making the server's ports untraceable by foreign hosts. This concept can be used by various applications as a means to improve security and avoid attacks.

The Software Defined Perimeter (SDP), also called Black Cloud, utilizes the SPA protocol in order to provide access to services, by protecting the system from attacks such as server scanning, denial of service, operating system and application vulnerability exploits, and man-in-the-middle attacks (assuming the attacker is not already inside the system).

The Cloud Security Alliance (CSA) has published a software specification document upon which this implementation was based. The CSA's proposal is similar to a DMZ(demilitarized zone): it isolates one or more services behind a perimeter (the SDP) while clients can access and gain access to services only through the gateway's of the SDP (as shown on figure 4.1).

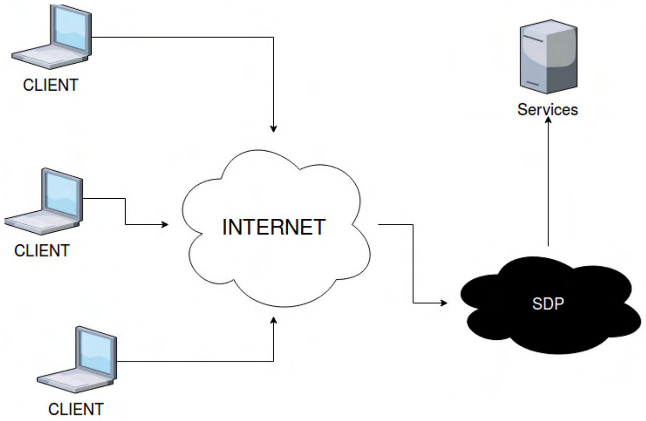

Figure 4.1: SDP protecting a service

The architecture and functionality of SDP can provide multiple applications such as: (i) **Enterprise Application Isolation**: Highly valuable enterprise applications can be isolated behind a Black Cloud while allowing authorized clients to access them. (ii) **Cloud Technology**: Multiple clients can be connected through a cloud which uses an SDP to provide various services to each client dynamically. (iii) **Internet of Things**: Everyday objects could be connected to an SDP in order to secure data aggregation, component registration and other actions.

In the following sections the SDP protocol will be presented as well as its implementation which was created during this thesis.

4.1 Architecture

SDP consists of two components: the SDP Controller (sCTL) and the SDP Hosts. An SDP host can either be an Initiating Host (IH) or an Accepting Host(AH).

The SDP Controller (sCTL) manages and authenticates SDP hosts while it also decides which of them communicate with each other.

IH represents the client entity. The IH can request access and communicate with services via the SDP system.

The SDP uses AH as gateways to services (thus they are also refered to as SDP Gateways). Their function is similar to a reverse-proxy: they redirect data received from clients to the corresponding services and then forward the services responses back to the clients. The phrase 'AH protects a service' is used to symbolize that an AH is configured to act as a gateway for a service.

The AH connects and stays connected to an sCTL until its termination. The IH connects to the sCTL in order to search for available services. When the IH selects the service it wants to use, the sCTL instructs it to connect to the corresponding AH which provides these services. The full architecture can be seen on Figure 4.2.

Both AH and sCTL use the SPA, developed through this thesis, in order to authenticate their clients. This means that once started, both hosts will configure their firewall with a strict DROP-all-packets rule. Later on, firewall access will be provided only to authenticated clients.

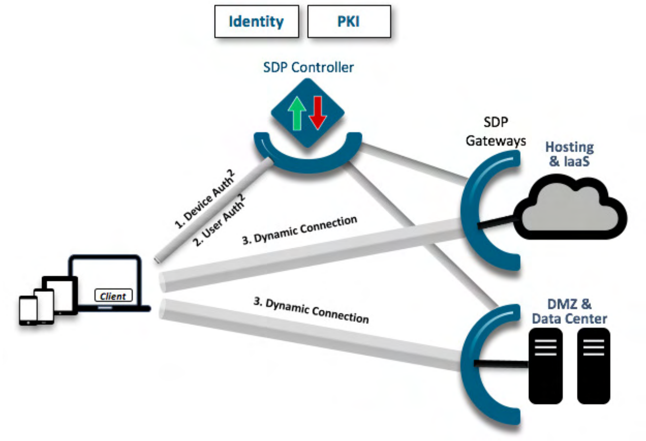

Figure 4.2: SDP architecture as illustrated on CSA's whitepaper

4.2 The SDP Protocol

The SDP protocol defines the authentication, communication and structure of the hosts in the SDP system (IH, AH, sCTL). The main components of this protocol are:
(i) **AH initialization**: Where AH connects to the sCTL and acquires information about the services it is going to protect. (ii) **IH initialization**: Where IH connects to the sCTL and acquires information regarding the available services in the SDP system. (iii) **AH-IH connection**: Where IH connects to an AH. (iv) **Dynamic tunnel mode**: Where an IH uses the AH as a gateway to a service.

The following subsections provide insight to each one of these steps. In the last subsection the changes made to the original SDP protocol are discussed.

4.2.1 SDP Authentication

In the SDP protocol the sCTL serve IH and AH entities, and AH entities serve IH entities. An entity shall be considered as an SDP server if it provides services to other entities, therefore since the sCTL serves IH and AH entities is an SDP server. An SDP client is an entity that is provided services by an SDP server, thus IH is an SDP client. Since AH provides services to IH entities and requests services from the sCTL it is considered both an SDP client (when dealing with the sCTL) and an SDP server (when serving an IH). The SDP Servers use the SPA protocol as an authentication mechanism, this means that they run an SPA server as a background daemon thread (therefore access to all ports is initially blocked by the SPA server's firewall configuration). Once the SDP client authenticates itself to the SDP Server they exchange messages over SSL connections.

The SDP uses a 2-step authentication mechanism. In the first step the SDP client needs to authenticate to the SPA server, running as a daemon thread on the SDP server, in order to gain firewall access to the SDP Server. On the second step the SDP client needs to authenticate itself to the SDP server as well. This process is illustrated on figure 4.3.

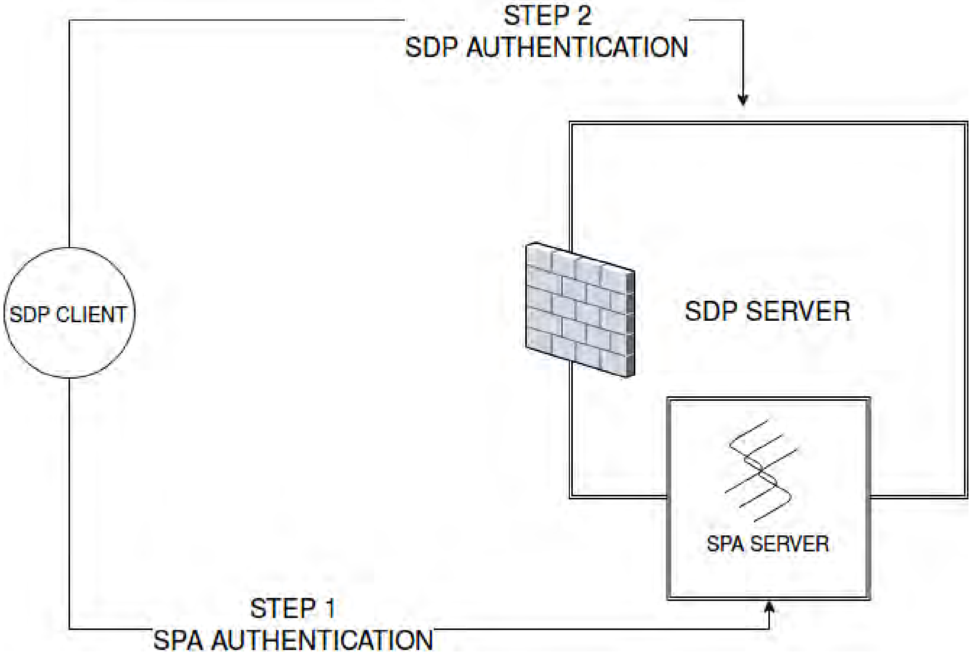

Figure 4.3: SDP's 2-step authentication mechanism

Once the SDP client is authenticated to the SPA server and gains firewall access to the SDP server, it has to authenticate itself once again. In order to authenticate to the SDP server the SDP client must first connect to the SDP server, through an SSL connection and send a LOGIN or OPEN_CON_REQ packet. The LOGIN packet is used for IH-sCTL and AH-sCTL communication while the OPEN_CON_REQ for IH-AH.

Since the SDP uses the SPA protocol, its hosts are also identified by the identification value that the SPA protocol uses, the AID value. For this reason the LOGIN and the OPEN_CON_REQ packets contain the AID of the SDP client that wants to login to the SDP server and another field called E_AID which is the result of encrypting the client's AID value by using the secret seed that this client shares with the SPA server. The SDP client is considered authenticated to the SDP server if the decryption of the E_AID value produces the AID. If an SDP client is authenticated the SDP server responds with an LOGIN_RESP (response to the LOGIN packet) or a OPEN_CON_RESP (response to the OPEN_CON_REQ packet). On the other hand, if SDP Client fails to authenticate itself, the SDP Server terminates their connection. The entire process is depicted on figure 4.4.

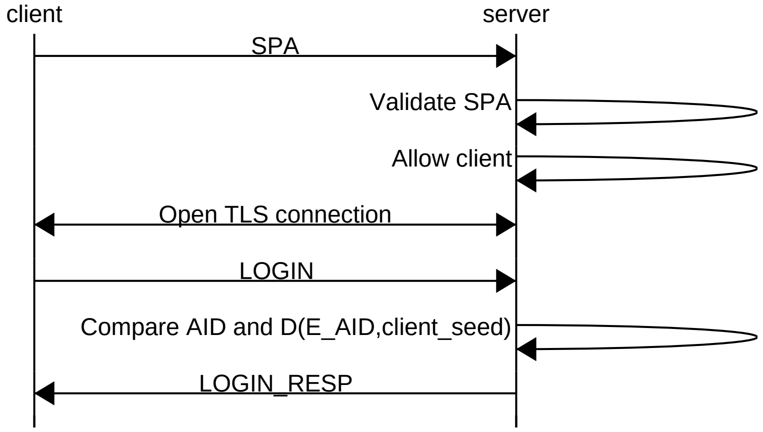

Figure 4.4: SDP Client authenticates to SDP Server

4.2.2 AH initialization

Once an AH has been initiated it needs to connect with an sCTL. The first step in this proccess is to authenticate to the sCTL's SPA server by sending an SPAp. Afterwards the AH sends a LOGIN packet to authenticate to the sCTL and complete the 2-step authentication proccess. Once the AH has been authenticated, sCTL will respond with an LOGIN_RESP.

Upon authentication of the AH, sCTL checks its configuration files and finds out which services must be provided by this AH. Afterwards the sCTL notifies the AH about these services by sending it a AH_SERVICES packet. An AH_SERVICES packet contains a list of services and information about each one of them such as : service type, IP, port, name and the unique ID of the service.

After receiving the AH_SERVICES packet, the AH is ready to act as a gateway to each one of the services included in this packet. The proccess is depicted on figure 4.5.

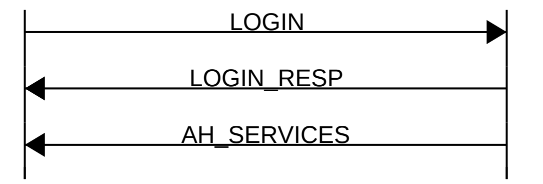

Figure 4.5: AH initialization

4.2.3 IH initialization

The IH is the representation of the client of the SDP system. Its first action is to authenticate to the sCTL via the 2-step authentication proccess (SPA and LOGIN).

Afterwards the IH can perform a service query via an IH_QUERY packet. The information included in IH_QUERY packets consists of the fields 'name', 'type' and 'service_id'. This packet can be used to find information about available services matching a specific query (by setting the name and type fields of the packet). Once the sCTL receives an IH_QUERY packet, which requests service-specific information, it responds with an IH_SERVICES packet, which contains information regarding the services requested such as their name, type and unique IDs. Another use of the IH_QUERY packet is to explicitly ask for a connection to a specific service by setting the service_id field of the packet to the ID of the corresponding service. Upon receival of such a packet, the sCTL searches for an available AH that is able to provide the service to the IH. It later sends an IH_AUTH packet to this AH to inform it about the IH request to connect with one of the services it protects, this packet contains the AID of the IH, the ID of the requested service and a seed to be used in order to decrypt and encrypt data transmitted during the authentication of this IH. The AH prepares for the connection and notifies the sCTL that it is ready to accept the client with an IHA_ACK. This packet includes the port which will be opened for the IH after it completes its authentication to the SPA server.

Finally, the sCTL notifies the IH with an AH_READY packet about the AH that is going to be used as a gateway to the requested service. Information such as the IP and port of the SDP server running on AH as well as the seed to use during its authentication to the AH are included in this packet. When this packet is received by the IH the communication with the AH can finally begin. The work of the sCTL is finished so it terminates the connection with the IH. Figure 4.6 displays this interaction.

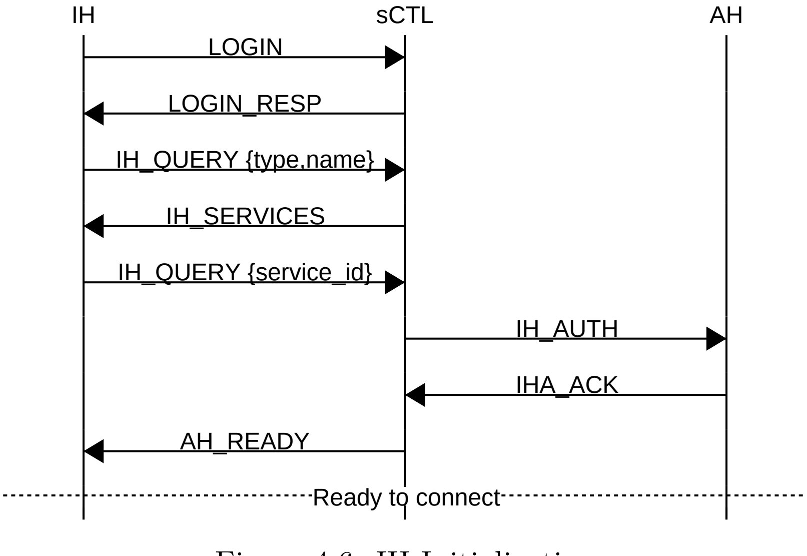

Figure 4.6: IH Initialization

4.2.4 IH-AH connection/DTM mode

After the IH initialization an AH is ready to accept connections from this IH. However the IH still needs to perform the 2-step authentication in order to authenticate itself to the AH (in order to prevent session hijacking). Therefore the IH authenticates itself via SPA to the SPA server background thread of the AH and later on, it sends an OPEN_CON_REQ to authenticate itself to the AH. Througout the 2-step authentication the IH uses the seed sent to it by the sCTL in the AH_READY packet. Once authenticated the AH will initiate a connection with the service that this IH wants to use.

The last and final component of the SDP protocol is the Dynamic Tunnel Mode (DTM). By using the AH as a gateway to the requested service, the IH sends to the service data encapsulated inside SDP packets called DATA packets. The AH will later unpack the SDP packet and send its payload to the service, while also forwarding its response back to the IH. When the IH has finished communicating with the services it sends an CONN_CLOSE packet to the AH to terminate the connection. The entire proccess can be seen at figure 4.7.

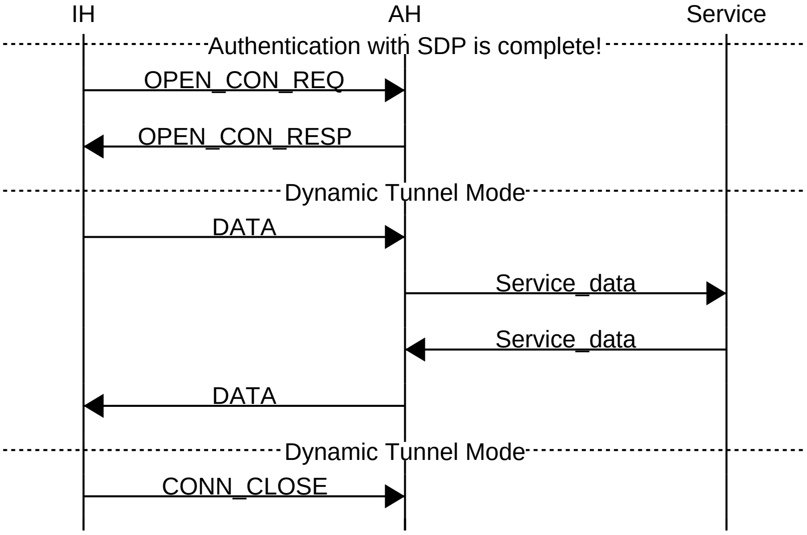

Figure 4.7: IH-AH connection and the Dynamic Tunnel Mode

4.2.5 Changes to the original protocol

Two major changes have been made to the original protocol proposed by CSA. The 2-step authentication and the addition of the IH_QUERY and the modification of the IH_SERVICES packets. Both will be explained below.

2-Step authentication

The inital protocol did not refer to the SDP server authentication. It stated that the basic authentication is completed once the SDP client has been authenticated to the corresponding SPA server. Also the LOGIN packet did not contain any information. However, once an SDP client has been authenticated to an SPA server and has gained firewall access to the SDP server, an attacker could use IP spoofing to hijack its session. In order to protect the system against session hijacking the extra authentication step between the SDP client and the SDP server was introduced.

The IH_QUERY-IH_SERVICES Packets

In the original protocol the IH_QUERY did not exist and the IH_SERVICES had a different form. The SDP specification stated that once an IH has been logged in to the sCTL, the sCTL would have to notify every available AH about it via an IH_AUTH. Thus, every AH would accept requests from this user. Also, the IH_SERVICES packet sent to the IH would contain a list of every available service as well as a list of AH entities that provide it.

However, in this implementation, the IH should explicitly select the service it is going to use and then the sCTL selects an AH to serve it. This architecture was selected because it minimizes the number of exchanged messages and the amount of information which every AH needs to store. In a real life scenario, where an sCTL would have been connected with multiple AH nodes, each time an IH would connect to the sCTL, IH_AUTH packets would have to be sent from the sCTL to every connected AH. Obviously the number of IH would be much greater than the number of AH so that means that this proccess would be repeated very often. Assuming that there areN_{AH} AHs in the SDP system, a total of $N_{AH} + 1$ packets would have to be sent for every logged in IH (where the final packet is the IH_SERVICES). The current implementation sends only four to six packets regardless of the number of AH in the system. Furthermore, the size of the IH_SERVICES packet in the old implementation would also increase over time as more AH are added to the system.

4.3 Implementation

During this thesis an implementation for SDP was created. This implementation, called *sdp_proto*, is available for use. In this section various implementation choices, made while developing the libraries for the SDP system, will be discussed.

4.3.1 The SDP packet

The standard SDP packet is displayed on table 4.1 When the packet is received, the receiver handles the data field based on the OP value. The OP value signifies the operation to be performed, available operations are LOGIN, LOGIN_RESP, IH_SERVICES, etc.

Table 4.1: SDP Packet Structure

Name	Size	Position	Description
OP	1B	b0	Operation to be performed
LENGTH	4B	b1 - b4	Length of Payload
DATA	LENGTH	b5-b(LENGTH+4)	The Payload

For a detailed list of the SDP packets used in this system one needs to refer to Appendix C.

4.3.2 Communication between hosts

The hosts use SSL connections in order to communicate with each other. So far only the AH and the sCTL run an SSL server, since they are SDP servers. In this implementation, to save resources and time, all SSL servers (sCTL and connected AHs) use the same pair of private and public keys. In future implementations AH SSL servers could use a different private key which would be obtained by the IH via the AH_READY packet.

4.3.3 ACK messages

Even though SSL, provided in the original SDP protocol, offer an efficient and reliable network-level service, an ACK system for the application layer was also required for the new implementation. SDP did not include instructions or details on ACK packages. The figure 4.8 displays the ACK messages scheme.

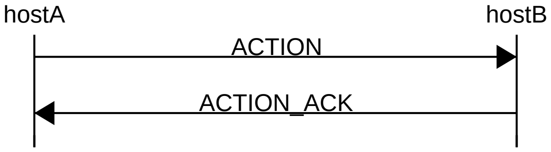

Figure 4.8: Simple ACK scheme

The ACK packets are (i) **AHS_ACK**: Sent by the AH to the sCTL to indicate that the lists of services has been received. (ii) **IHA_ACK**: Sent by the AH to the sCTL to indicate that it is ready to connect with an IH. (iii) **LOG_ACK**: Sent by an SDP client to an SDP server when the SDP authentication is complete. Detailed information on ACK packets is provided in Appendix C.

4.3.4 Preventing DoS in the SDP system

It was mentioned in subsection 3.4.2 that a DoS attack is possible on SPA after the SPA server authentication with the SPA client. Since the firewall allows connections from requests originating from the SPA client's IP, an experienced attacker could spoof his/her IP and start a DoS attack on the open port. This is displayed on figure 4.9.

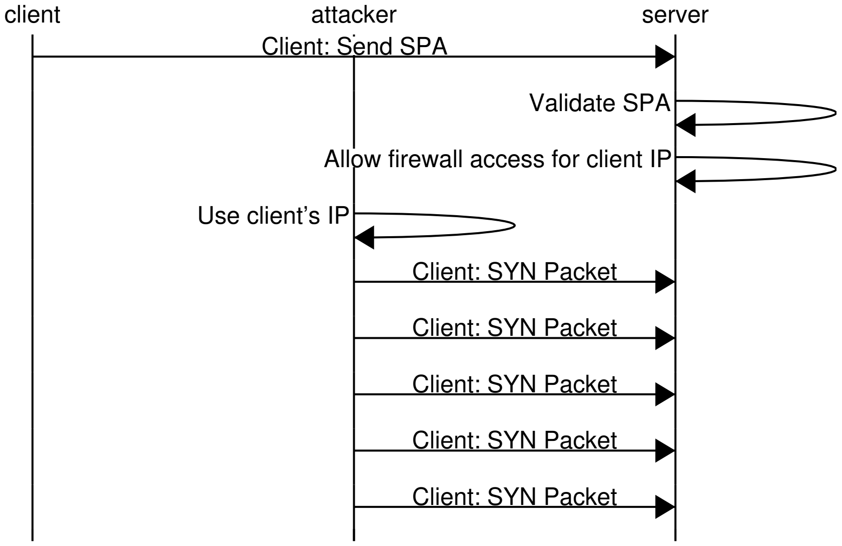

Figure 4.9: SYN attack on SPA mechanism

Each firewall rule that allows SPA client connections, matches new or established connections. The SDP system stops DoS attacks by setting this rule to match only established connections once the SDP Client has successfully logged in to the SDP Server (by using the *set_rule_to_established* function of the SPAListener class from *spa_lib*). Attackers are not able to authenticate themselves to the SDP system, due to the 2-step authentication mechanism, and during each failed attempt the SDP Server would terminate their connection. Eventually, once the SDP client authenticates itself, the SDP Server will block access to new connections originated by its IP and only the established connection of this client will be preserved. This is displayed on figure 4.10.

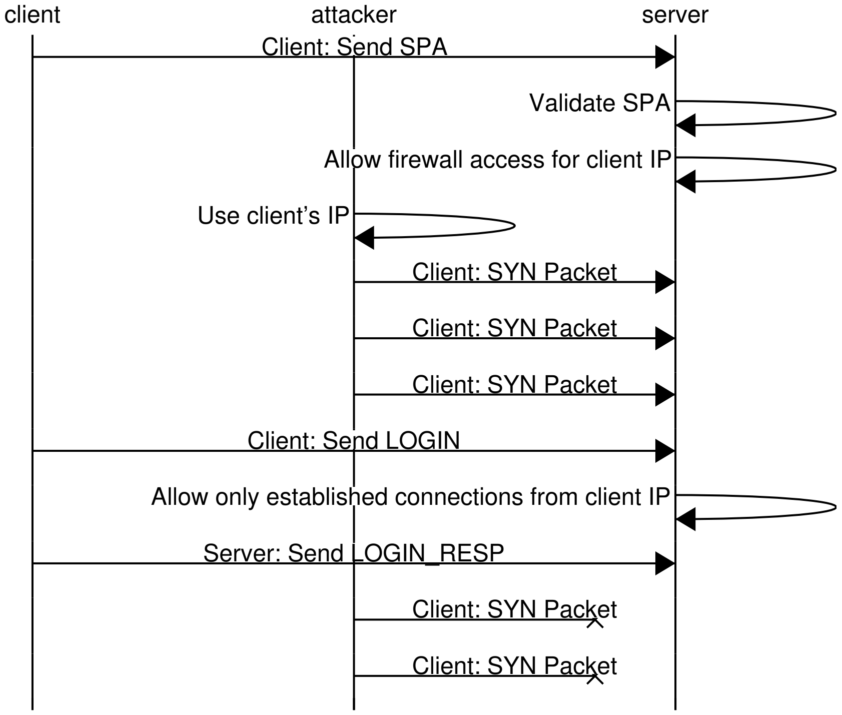

Figure 4.10: Preventing SPA-DoS attacks on SDP

In the scenario where the SDP client does not authenticate itself to the SDP server after the SPA authentication, the firewall will still accept connections from the client's IP and the DoS attack would be possible until the client logs in. In this implementation it was assumed that the SDP client will always try to authenticate itself after the SPA packet so no precautions were taken to solve this problem. However, creating a background daemon thread, which blocks access to the firewall if a connection has not been made after a specific period of time, would easily solve the problem.

4.3.5 SDP Library

The SDP Library developed in this thesis exports the following classes/modules : *sdp_IH*, *sdp_AH* and *sdp_CTL*. An analytical description of the API is provided in the Appendix A and B. The scripts named startCtl.py and startAH.py, use the *sdp_AH* and *sdp_CTL* APIs to set up and run the sCTL and the AH respectively.

Whenever an application wants to use the SDP system it has to use the *sdp_IH* API. The *sdp_IH* acts as a middleware. It logs into the SDP system by using the given IH-client credentials (such as AID, PASSWORD and SEED), before forwarding application packets to services. In the following pages, an application using the *sdp_IH* module will be considered as an IH itself.

The API of the emphsdp_proto library is given in Appendix B.

Chapter 5

Proof of concept

5.1 The Ping-Pong Application

In order to test the SDP library, a simple 'ping pong' application was developed. The client sends TCP packets containing the message 'Ping!' as a payload, while a service responds with 'Pong!' whenever such a packet is received. Since it was necessary to demonstrate the SDP implementation's usability and features, only two machines were used. One of them hosted the Controller, the AH and the service (machine2-192.168.0.97), while the other hosted the Ping application (machine1-192.168.0.106). The client sending the Ping message (hereafter referred to as the Ping application) uses the *sdp_IH* to forward the packets, through the SDP system, to the service which responds with a Pong (hereafter referred to as the Pong service). Figure 5.1 depicts the setup.

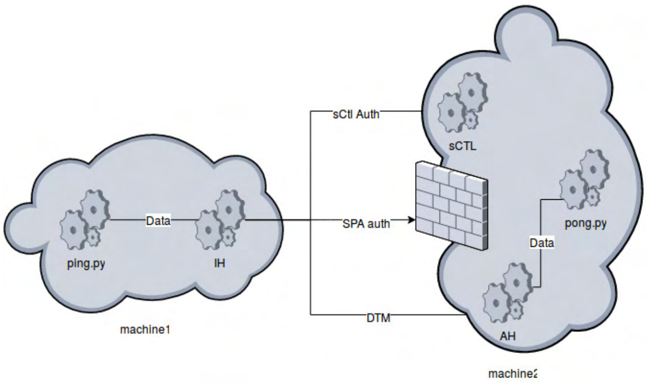

Figure 5.1: Testing enviroment

5.1.1 Controller - Accepting Host

When the SDP Controller is initialized, it immediately configures the firewall of the hosting machine to drop all incoming connections. In order to check the INPUT chain for the firewall the *iptables* [25] linux command was used.

```
machine2@VM_sCTL#iptables -L INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere
```

After the sCTL's initialization, the Pong Service and the AH are activated. The AH is situated in the same network as the sCTL (therefore the sCTL considers the AH as localhost). By initializing the connection and providing a valid SPA packet, the AH gets the Controller to configure the firewall settings such that the AH is allowed to communicate with the Controller.

```
machine2@VM_sCTL#iptables -L INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  localhost            anywhere        cstate NEW, ESTABLISHED
DROP       all  --  anywhere             anywhere
```

It was expected that, since the AH also initializes its own firewall and blocks access to all its ports, it would have inserted a new DROP all rule into the INPUT chain. However, the AH, before blocking incoming connections, checks if the sCTL is running on its local system. If it is, then the AH does not add the DROP rule since the SPAs daemon running on the sCTL has already inserted it.

As discusssed in section 4.1.1, after this authentication phase, a new session is created for the AH, and then the AH receives a list of the services that it is going to provide from the sCTL.

```
machine2@VM_sCTL#python startCtl.py
2018-01-24 09:56:36,453 - SDP_access - WARNING - Started TLS server on
 port 7000
2018-01-24 09:56:36,454 - SDP_access - WARNING - Started SPA server
2018-01-24 09:56:43,090 - SDP_access - INFO - Got new connection from
 address : 127.0.0.1
2018-01-24 09:56:43,105 - SDP_access - INFO - New login request from
 client :e83bf2f40b4b11e7932a48e2440c8ced. Creating new AH session
2018-01-24 09:56:43,108 - SDP_access - DEBUG - Sending services to AH
 with aid : e83bf2f40b4b11e7932a48e2440c8ced
```

```
machine2@VM_AH#python startAH.py
2018-01-24 09:56:43,093 - AHost_access - WARNING - Connected with Controller.
 Starting TLS Connection
2018-01-24 09:56:43,106 - AHost_access - INFO - Connected with server.
 Session ID : e189e258-00ec-11e8-8438-646e69b26c4f
2018-01-24 09:56:43,110 - AHost_access - INFO - Updated list of services
```

5.1.2 Controller - Initiating Host

As mentioned in the previous section 3.1, the first step of every attacker is to scan a system and try to acquire as much information as possible from it. By using *nmap* to scan the machine2 system from the machine1 terminal, it is proven that it is impossible to gain information by port scanning:

```
machine1@VM_IH#nmap -O 192.168.0.97

Starting Nmap 7.01 ( https://nmap.org ) at 2018-01-22 15:30 PST
Nmap scan report for 192.168.0.97
Host is up (0.0011s latency).
```

```
All 1000 scanned ports on 192.168.0.97 are filtered
Too many fingerprints match this host to give specific OS details

OS detection performed. Please report any incorrect results at
 https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 15.52 seconds
```

As expected, machine2, although detectable in the network, is impervious to analysis.

The next step for the Ping application is to use the *sdp_IH* and connect with the sCTL. this will start a chain of events:

First, after succesful authentication, a new entry is added on the protected machine's firewall.

```
machine2@VM_sCTL#iptables -L INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  192.168.0.106        anywhere           cstate NEW, ESTABLISHED
ACCEPT     tcp  --  localhost            anywhere           cstate NEW, ESTABLISHED
DROP       all  --  anywhere             anywhere
```

Afterwards, the IH initiates a TCP connection with the controller. The Ping application/IH performs a query asking for the Pong Service. The sCTL notifies the available AH, exchanges client information and forwards information about the available AH to the IH. Then, it terminates the connection with the IH.

```
machine2@VM_sCTL#python startCtl.py
.
.
.
2018-01-24 09:57:16,659 - SDP_access - INFO - Got new connection from
 address: 192.168.0.106
2018-01-24 09:57:16,675 - SDP_access - INFO - New login request from
 client : e83bf2f40b4b11e7932a48e2440c8ceC. Creating new IH session
2018-01-24 09:57:17,669 - SDP_access - WARNING - [!] Request from user
 e83bf2f40b4b11e7932a48e2440c8ceC  to connect with AH
 e83bf2f40b4b11e7932a48e2440c8ced
2018-01-24 09:57:18,205 - SDP_error - ERROR - Session :
 f58bea3a-00ec-11e8-8438-646e69b26c4f terminated by host
```

The final step is the data exchange between the Ping application and the Pong service. The AH, after receiving the IH_AUTH packet sent by the sCTL, can authenticate SPA packets sent by the IH. When the SPA authentication is completed, the IH logs in to the AH, performs the data exchange and terminates its connection to the SDP system.

```
machine2@VM_AH#python startAH.py
.
.
.
2018-01-24 09:57:18,152 - AHost_access - WARNING - New client with aid
 e83bf2f40b4b11e7932a48e2440c8ceC
---Seed: 2DIFXTTKGKQUAOF4H19DNJQQH27W1BGL
---Service: 2
```

```
2018-01-24 09:57:18,153 - AHost_access - WARNING - Started IH SSL Listener
 on port 6767
2018-01-24 09:57:18,570 - AHost_access - WARNING - Got new connection from
 address 192.168.0.106
2018-01-24 09:57:18,630 - AHost_access - WARNING - Got CONN_OPEN message
 from user with aid e83bf2f40b4b11e7932a48e2440c8ceC
```

```
machine1@VM_IH#python ping.py --sdp
2018-01-24 09:57:16,679 - IHost_access - INFO - Connected with server.
 Session ID : f58bea3a-00ec-11e8-8438-646e69b26c4f
2018-01-24 09:57:18,161 - IHost_access - INFO - AH is available at
 address : 192.168.0.97:6767
2018-01-24 09:57:18,205 - IHost_access - WARNING - Got instruction to
 terminate Controller connection
2018-01-24 09:57:18,205 - IHost_access - WARNING - Listener is terminated
2018-01-24 09:57:18,748 - IHost_access - INFO - Authenticated with AH for
 service 2
-----Starting send Sequence
-----Ending send Sequence
2018-01-24 09:57:19,254 - IHost_access - WARNING - [!] Got Instruction to
 terminate IH
```

The output of the Pong Service is presented below:

```
machine2@VM_AH#python pong.py --tcp
Accepted TCP client at address ('127.0.0.1', 41886)
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
Got ' PING! 'from client 192.168.0.97
Sent 'Pong' message
```

Once authenticated in the SDP system, machine1 can finally use *nmap* to scan the protected machine and acquire more information about the host.

```
machine1@VM_IH#nmap -O 192.168.0.97

Starting Nmap 7.01 ( https://nmap.org ) at 2018-01-22 15:14 PST
Nmap scan report for 192.168.0.97
```

```
Host is up (0.13s latency).
Not shown: 997 closed ports
PORT     STATE    SERVICE
80/tcp   open     http
443/tcp  open     https
514/tcp  filtered shell
Device type: WAP|general purpose|storage-misc
Running (JUST GUESSING): Actiontec embedded (99%), Linux 3.X|2.4.X (99%),
Microsoft Windows 7|2012|XP (96%), BlueArc embedded (91%)
OS CPE: cpe:/h:actiontec:mi424wr-gen3i cpe:/o:linux:linux_kernel cpe:/o:
 linux:linux_kernel:3.2 cpe:/o:linux:linux_kernel:2.4.37 cpe:/o:microsoft
 :windows_7 cpe:/o:microsoft:windows_server_2012 cpe:/o:microsoft:windows_xp
 ::sp3 cpe:/h:bluearc:titan_2100
Aggressive OS guesses: Actiontec MI424WR-GEN3I WAP (99%), Linux 3.2 (98%),
 DD-WRT v24-sp2 (Linux 2.4.37) (97%), Microsoft Windows 7 or Windows
Server 2012 (96%), Microsoft Windows XP SP3 (96%), BlueArc Titan 210
 0 NAS device (91%)
No exact OS matches for host (test conditions non-ideal).

OS detection performed. Please report any incorrect results
 at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 44.63 seconds
```

5.2 DoS attack scenario

In the second and final section of this chapter, SDP is tested against one of the threats it claims to be most efficient against, DoS attacks. The attack was executed using *codingplanets* tool *Overload-DoS*[3] which can be found on github.

One of the most prominent DoS attacks is SYN-ACK attack. The attack is famous due to its simplicity and effectiveness. The core concept is exploiting the TCP handshake mechanism. TCP uses a handshake in order to initialize a connection. During this handshake three messages are exchanged:

1. SYN : Sent by the client to the server.
2. SYN-ACK : Sent by the server to the client to acknowledge the SYN packet.
3. ACK : Sent by the client to the server to acknowledge the transaction

During those steps, both parties exchange information between them such as the sequence numbers and window sizes which are going to be used throughout the communication.

As soon as the SYN packet is received by the server, it initializes a connection and waits for its completion. If the ACK packet is not received after a predefined period of time, the server terminates the connection. Thus, the server is allocating resources, even if the connection has not properly started.

SYN attacks, or half-open attacks, use this knowledge to increase server load to the point which, it will not be able to provide services to new clients. this attack, sends multiple SYN packets, without responding with the ACK packet when the SYN-ACK is received. In this way the server uses resources to initialize connections and waits for pending ACK packets, while the SYN requests multiply. Figure 5.2 displays the attack.

The attack will be demonstrated using machine1 and machine2 from section 4.4. The first step, is to use the Overload-DoS script, on machine1(192.168.0.106), to initiate a DoS attack against the Pong service, which runs on machine2(192.168.0.97) and port 8080.

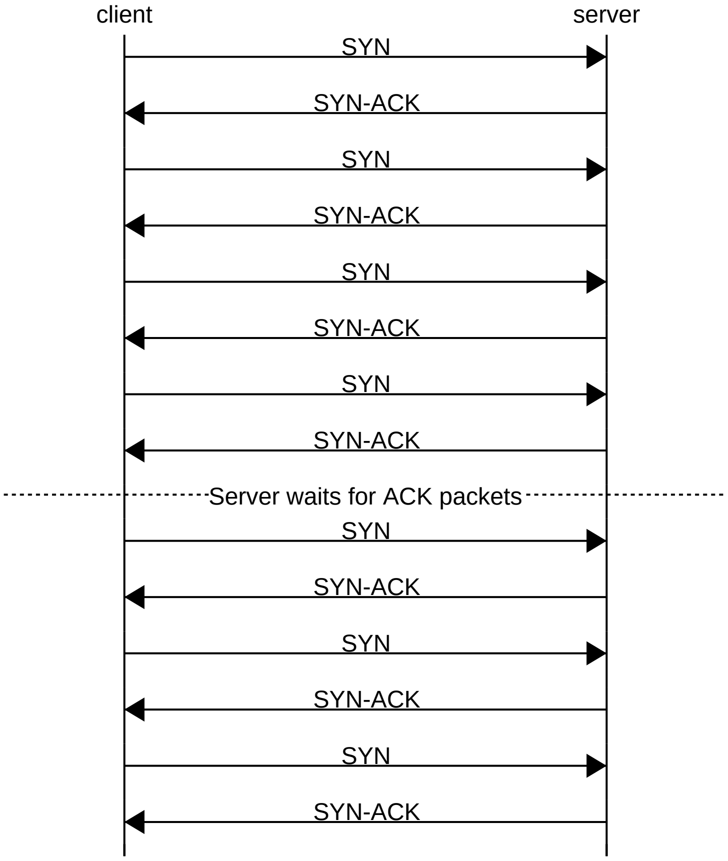

Figure 5.2: SYN-ATTACK example

```
machine1@attacker#python Overload-DoS.py -target 192.168.0.97 -port 8080\
 -threads 1 -syn
[*] You have enough permisson to run Overload-v1.0
[*] SYN flood started on: 192.168.0.97
```

The *-syn* option, instructs the program to perform a SYN attack on target 192.168.0.97 and port 8080, using only one thread. Every packet sent through this program, is sent using a different spoof ip address.

The program *netstat* [5] provides network connection information. During the attack, machine2 gets the following output using netstat with the options *-t*, in order to get the tcp connections, *-n* to get numerical addresses, *-p* to obtain program PID and *-a* for including listening and non listening sockets in the results.

```
machine2@victim#netstat -antp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:6379          0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:8080          0.0.0.0:*               LISTEN
18558/python
                                .
```

```
                                          .
                                          .
tcp6       0      0 127.0.0.1:8080          224.202.114.192:54321   SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          232.91.230.174:54321    SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          231.137.68.99:54321     SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          227.118.65.237:54321    SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          237.229.29.194:54321    SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          225.74.183.222:54321    SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          230.35.178.14:54321     SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          237.59.127.188:54321    SYN_RECV
-
tcp6       0      0 127.0.0.1:8080          232.52.24.245:54321     SYN_RECV
-
                                          .
                                          .
                                          .
```

The first thing to note here is that a lot of tcp sockets are on the SYN_RECV state, meaning that they have obtained a connect request (SYN packet) and they wait for the ACK message. It is also obvious that, despite being sent from the same machine, every request seems to have been made by a different IP. this is because Overload-DoS, uses IP spoofing, to hide the attacker's address in the network. Therefore, without the SDP protection, the system is vulnerable to DoS attacks.

In the second test to be performed, the Ping application was hosted alongside AH, sCTL and the Pong service on machine2. Ping Application (similarly to 4.4) was connected, via the IH API, to the SDP system and exchanged packets with the Pong service. machine1, on the other hand, was trying to perform a DoS attack on machine2, in the same port and address it used in its previous attempt.

```
machine1@attacker#python Overload-DoS.py -target 192.168.0.97 -port 8080\
 -threads 1 -syn
[*] You have enough permisson to run Overload-v1.0
[*] SYN flood started on: 192.168.0.97
```

However, this time, the SDP protects the hosts, and does not allow packets from unauthorized users to enter the SDP system. In order to prove its efficiency, *netstat* was once again used.

```
machine2@victim#netstat -atnp
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:6379          0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:6767            0.0.0.0:*               LISTEN
-
```

```
tcp        0      0 127.0.0.1:8080          0.0.0.0:*               LISTEN
5995/python
tcp        0      0 0.0.0.0:4369            0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:7000            0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:15672           0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:5432          0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:25672           0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:8080          127.0.0.1:37664         ESTABLISHED 5995/p
tcp        0      0 127.0.0.1:37664         127.0.0.1:8080          ESTABLISHED -
tcp        0      0 127.0.0.1:4369          127.0.0.1:55063         ESTABLISHED -
tcp        0      0 127.0.0.1:52966         127.0.0.1:7000          ESTABLISHED
```

It is obvious, that no SYN packets reached the Pong Application this time. The attack was unsuccessfull, since the SDP system blocked it, and the hosts are protected. Most importantly, communication can still take place inside the system, while the clients and the service are protected.

This example further proves the CSA claims about the efficiency of the SDP.

Chapter 6

The hSDP module

6.1 Motivation

The hSDP was developed in order to demonstrate the functionality of the SDP protocol for the case of browsing. The 'h' in its name stands for the HTTP protocol[17], which is used by web applications to access and interact with online content.

In practice, the hSDP is a module, used by the IH and AH libraries, which modifies the SDP packet's payload before sending it or after its receipt, when dealing with HTTP connections.

HTTP is an insecure service, which does not encrypt the transmitted data. By encapsulating HTTP packets inside SDP packets the hSDP provides to the HTTP protocol security mechanisms that were previously missing from it such as encryption and client authentication (similar to HTTPS[30]). A service (such as a website) would be benefited by using such a mechanism as it would have the SDP system acting as a reverse-proxy. Network traffic would be redirected to the AHs of the system before being forwarded to the actual website. Therefore, DoS, and other types of attacks, would not affect directly the website servers but the intermediate network nodes (AHs of the SDP system).

6.2 The functionality of hSDP

HTTP requests are series of raw text, which describe the request and its options. A typical HTTP request is provided bellow:

```
GET / HTTP/1.1
Host: www.e-ce.uth.gr
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (X11; Linux x86_64)
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Cookie: _ga=GA1.2.1074060408.1508268236
```

A response to this request would be :

```
HTTP/1.1 200 OK
Date: Sun, 03 Dec 2017 20:35:45 GMT
Server: Apache
Content-Encoding: gzip
Vary: Accept-Encoding
Keep-Alive: timeout=5, max=88
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
<RESPONSE DATA>
```

hSDP's primary aim is to encapsulate these requests into SDP packets.

When the IH sends DATA packets to an HTTP service, it uses the hSDP module to handle them. When these packets are received from the AH they are once again handled by the hSDP module. Each HTTP request that needs to be sent through the SDP system must be provided along with an unique identification number (ID) by the program that uses the *sdp_IH*. The IH, with the use of the hSDP module, links this request with the given ID and later dissassembles it to smaller packets (depending on the request's initial size). Each of this packets is inserted into a seperate DATA packet (DATA packets encapsulating HTTP payloads are called hSDP packets). The packets are then sent to the appropriate AH.

Once the AH gathers hSDP packets, it assembles them together, by using the hSDP, to form the inital HTTP request which is later forwarded to the HTTP service. Upon the receipt of the HTTP response, the AH once again uses the hSDP module to split it into hSDP packets and forwards them to the IH. The IH collects the hSDP packets and creates the HTTP response. The response is returned to the program along with the ID that the corresponding HTTP request was provided with.

6.3 hSDP packet structure

The hSDP packets are illustrated in table 6.1. The field serial_no reffers to the serial number of this packet, which specifies its position in the hSDP packet set. The field con_id is the unique identification number for the HTTP REQUEST that this hSDP packet belongs to while is_last field contains a boolean value which is set to true in case the packet is the last in the set.

Table 6.1: hSDP DATA payload

Name	Field
data	The actual HTTP payload
con_id	The unique ID for the connection
serial_no	The serial number of the payload
is_last	If this is the final packet in the request

6.4 Testing hSDP

In order to test the mechanism two different scripts were created, *ih_proxy* and *ah_proxy*. A browser application, is able to send requests and receive responses through the SDP system, by redirecting its HTTP REQUEST packets to the local *ih_proxy* instance (by TCP send or another connection type such as TLS/SSL).

The *ih_proxy* script, uses the IH API to forward its packets through the SDP system while the *ah_proxy* is a script which runs as a service. Its main functionality is to act as a proxy and forward incoming HTTP packets to their destinated services and then redirect the responses back to the clients that made the requests.

6.4.1 Implementation

In this implementation, the AH protects the *ah_proxy* service, which means that IHs can access this service through it. The Chromium Browser was used to generate network traffic which was later redirected to the *ih_proxy*. The process, which can be seen on figure 6.1, is defined bellow.

Algorithm 4: Sending packets through hSDP

1: The browser creates an HTTP request and forwards it to the *ih_proxy*.
2: *ih_proxy* receives the incoming traffic. Using the IH API, it encapsulates the packets received into hSDP packets and then forwards them, using *send_data* function, to the AH which it is linked with.
3: AH gets the incoming packages from *ih_proxy*, parses them to an HTTP REQUEST packet and sends it to *ah_proxy*.
4: *ah_proxy* receives the REQUEST from the AH, forwards it to its actual destination (a website) and waits for its response. When the response is received it redirects it back to the AH.
5: AH gets the received HTTP RESPONSE from the *ah_proxy*, dissassembles it into hSDP packets and then forwards them to *ih_proxy*.
6: *ih_proxy* checks for new SDP packets, using the IH API's *get_data* function. When the packets are received from the AH, it assembles them to an HTTP Response and forwards it to the browser.
7: The browser uses the HTTP RESPONSE to display the content received from *ih_proxy*.

6.4.2 Usage Instructions

In order to run a full demo of the application one has to :

- Start the sCTL: *python startCtl.py*
- Start an AH: *python startAH.py*
- Start the *ah_proxy* service: *python* ah_proxy*.py*
- Start the *ih_proxy* client: *python ih_proxy.py*
- Set chromium browser to redirect HTTP requests to the ih_proxy: *chromium –proxy-server="http=127.0.0.1:8000;"*

- Visit an HTTP website on the Chromium browser

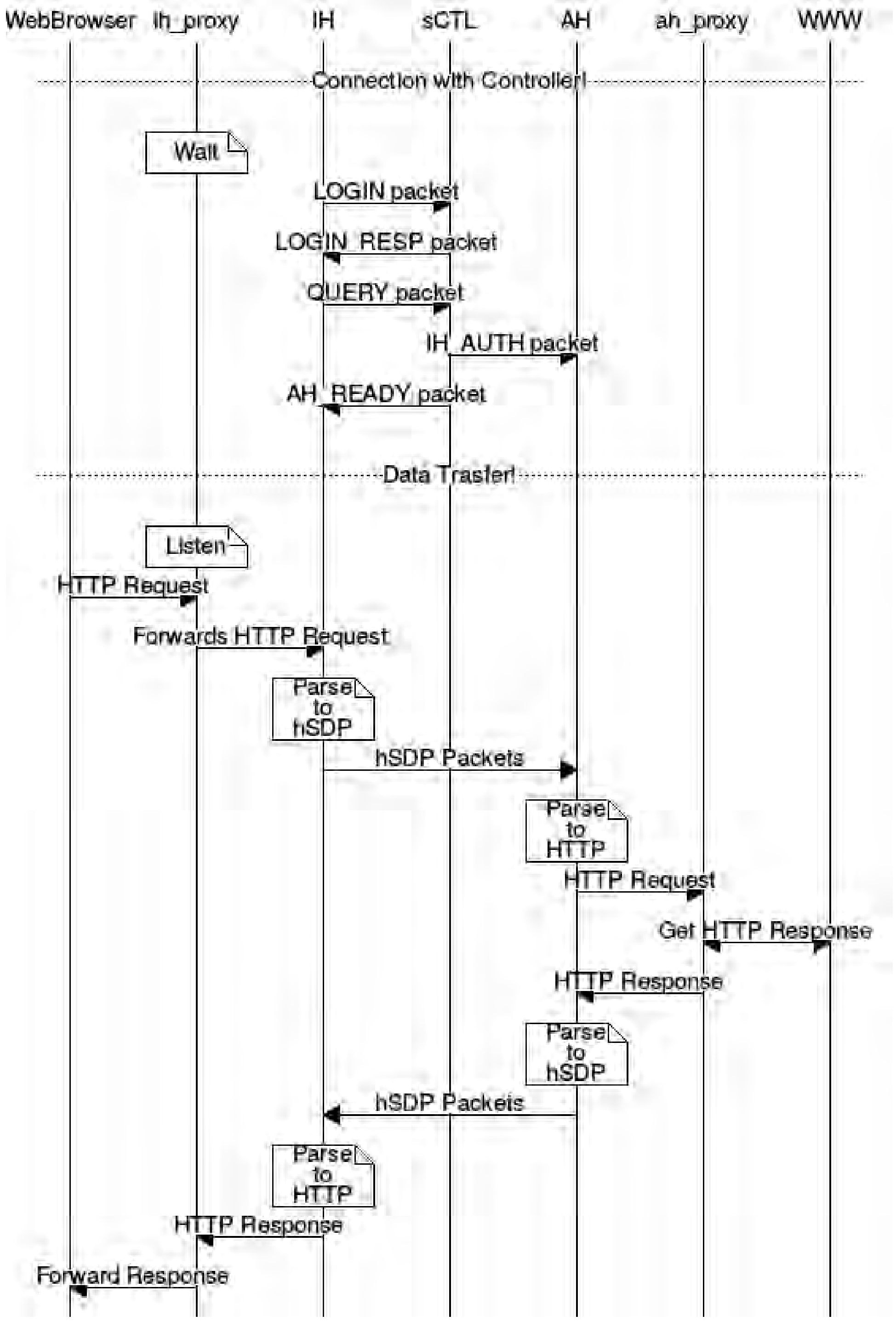

Figure 6.1: hSDP in action

Chapter 7

Complexity and Performance Evaluation

This chapter presents an evaluation of the SDP implementation. First, the complexity of the iplementation is reported in terms of lines of code (LOC). Then, an analytical model for estimating the overhead of SDP operations is presented, based on a breakdown of the most important overhead costs that can be measured on an individual basis. End-to-end measurements are also performed, and the results are compared with the predictions of the model.

7.1 Implementation size

During this thesis two python libraries were produced, *spa* and *sdp_proto*. The *cloc* program was used to calculate the total number of lines for each of those projects. The results can be seen on table 7.1.

Table 7.1: LOC for the projects created

Library	Language	LOC
spa	Python	464
sdp_proto	Python	3177

During implementation the following python libraries were used : *python-iptables*, *pycrypto*, *bs4*, *uuid*, *scapy*, *netifaces*, *config*, *pyyaml*.

The source files require 190kB of space.

7.2 Performance model

To assess the overhead of SDP without having to perform end-to-end measurements that require a more complex tesbed setup, we construct an analytical model that takes into account simpler/fundamental cost factors that can be measured individually using a much simpler setup.

The basic cost variables of the proposed model are summarized in Table 7.2.

The cost of simple operations (such as additions, multiplications, loop and if statements) were not taken into consideration as they do not produce significant overhead.

Table 7.2: Cost Factors

UDP_{snd}	send UDP packet
SSL_{con}	setup a typical SSL connection
SSL_{snd}	send a message via SSL
CFG_r	read the configuration from a file
CFG_w	update configuration file contents
AES_e	encrypt data using AES algorithm
AES_d	decrypt data using AES algorithm
$MD5_m$	create md5 hash
FW_{add}	add firewall rule
FW_{rm}	remove firewall rule

The overhead of SDP is broken down into different components, which will be discussed in the following subsections.

7.2.1 SPA authentication

A basic operation in SDP is for the client to authenticate itself via SPA. The respective cost (delay) for this can be expressed as

$$SPA_{auth} = SPA_{auth_{client}} + UDP_{snd} + SPA_{auth_{server}}$$

where $SPA_{auth_{client}}$ is the time required for the client to prepare the SPA packet, UDP_{snd} is the cost for sending the SPA packet to the server, and $SPA_{auth_{server}}$ is the time required for the server to validate the packet and perform the necessary operation in order for the client to be able to connect to it.

In turn, $SPA_{auth_{client}}$ can be expressed as

$$SPA_{auth_{client}} = MD5_m + AES_e$$

where, $MD5_m$ is the cost for computing the MD5 hash of the SPA packet, and AES_e is the cost for encrypting the string with the AID, RANDOM, PASSWORD and NEW_SEED values in order to produce the ENC field of the SPA packet.

Also, $SPA_{auth_{server}}$ can be expressed as

$$SPA_{auth_{server}} = CFG_r + AES_d + MD5_m + CFG_w + FWadd$$

where CFG_r is the time required to read the configuration information for the sender (seed for this transaction, previous random values), AES_d is the time required to decrypt the ENC field of the SPA packet, $MD5_m$ is the time required to compute the MD5 hash of the packet, CFG_d is the time required to update/write the configuration for the sender, and FW_{add} is the time required to add a firewall rule in order to accept SSL connections from the client (if the authentication was successful).

7.2.2 SSL client authentication

Once the client is authenticated via SPA, it has to complete the authentication procedure by authenticating itself to the sever once again, this time via SSL. The respective cost can be expressed as

$$SSL_{auth} = SSL_{con} + AES_e + SSL_{snd} + CFG_r + AES_d + 2 * SSL_{snd}$$

where SSL_{con} is the cost for setting up the SSL connection, AES_e is the cost for the client to encrypt its AID that is stored in the E_AID of the LOGIN request, the first SSL_{snd} is the cost for sending the LOGIN request over the SLL connection, CFG_r is the cost for the server to read the configuration information for the connecting client, AES_d is the cost for decrypting the AID, and the last two SSL_{snd} is the cost for sending to the client a LOGIN_RESP to which the client responds (after initializing a session object) with an LOG_ACK so that the controller can do the same.

7.2.3 Total authentical cost

Thus, the total cost for a client to authenticate itself with a server is

$$SDP_{auth} = SPA_{auth} + SSL_{auth}$$

with SPA_{auth} and SSL_{auth} as already discussed above.

7.2.4 Service exchange cost

Once the SSL authentication has been completed, IH and AH both perform additional queries. AH to receive the services it is going to provide, and IH to connect with a service via an AH with estimated cost $SDP_{init_{AH}}$, $SDP_{init_{IH}}$ respectivelly. These actions are considered as initialization actions in the SDP system.

The $SDP_{init_{AH}}$ can be expressed as:

$$SDP_{init_{AH}} = SDP_{auth} + CFG_r + SSL_{snd} + CFG_w + FW_{add} + SSL_{snd}$$

where SDP_{auth} is the cost of the SDP authentication, CFG_r is the required cost for the controller to read the services AH provides from the configuration files, SSL_{snd} the cost of sending the AH_SERVICES packet, CFG_w the required cost for the AH to update service information on its configuration files, FW_{add} to add new firewall rules to allow communication with the services and SSL_{snd} is the cost of the final AHS_ACK packet send to the Controller.

Due to the complexity of the IH-AH authentication the overall cost has to be broken down in four different components

$$SDP_{init_{IH}} = SDP_{auth} + Q\_t + S_t + SDP_{authIH-AH}$$

where SDP_{auth} is the cost of the SDP authentication, Q_t is the cost of the query performed by the IH to get the available services, S_t the required cost for finding an available AH, and $SDP_{hand_{IH-AH}}$ the required cost to perform an SDP authentication with the AH.

The service query cost can be calculated as:

$$Q_t = 2 * SSL_{snd}$$

The first SSL_{snd} is the cost required from the IH to send the IH_QUERY packet and the second SSL_{snd} the cost required from the sCTL to respond with an IH_SERVICES packet.

The discovery of an AH has a total cost of :

$$S_t = SSL_{snd} + CFG_r + SSL\_snd + CFG_w + SSL\_snd + SSL_{snd}$$

$$S_t = 4 * SSL_{snd} + CFG_r + CFG_w$$

where the first SSL_{snd} is the time required to send the IH_SERVICES packet from the IH, to indicate the service it intends to use, CFG_r the cost required for the sCTL to read its configuration file in order to select an appropriate AH, the second SSL_{snd} is the required cost to notify the available AH via an IH_AUTH packet, CFG_w the cost required for the AH to change its configuration files, with information about the new IH, and the final two SSL_snd is the cost for sending the IHA_ACK from the AH to the sCTL and the AH_AUTH from the controller to the IH.

The $SDP_{authIH-AH}$ is equal to SDP_{auth} since the exact same actions are required in order to authenticate a client to the sCTL and an IH to an AH. Therefore

$$SDP_{auth} = SDP_{authIH-AH}$$

7.2.5 Data exchange

DTM, is the actions performed by the IH to exchange data with a service via an AH. Its cost can be calculated as

$$SDP_{serv} = SSL_{snd} + S_{proc} + SSL_{snd}$$

where the first SSL_snd is the required cost for sending the DATA packet from the IH to the AH, S_{proc} are the service related costs (processing info, sending and receiving to and from the AH respectivelly) and the final SSL_snd is the required time for forwarding the service response to the IH. (In calculating the prediction for the performance cost of the SDP_{serv}, S_{proc} will be consider minimal).

A complete communication between an IH and a service can be calculated as

$$SDP_{sev_{total}} = N * SDP_{serv}$$

where N is the number of packets sent.

7.2.6 Logout cost

In the end of the communication between two hosts a LOGOUT packet must be sent. Its cost is calculated as

$$SDP_{logout} = SSL_{snd} + FW_{rm}$$

where SSL_{snd} is the cost required to send the LOGOUT packet and FW_{rm} the cost required from one of the hosts to remove the other from its firewall. In IH-sCTL and AH- communication the sCTL is always the one to remove the client from the firewall, while in the IH-AH communication, the AH removes the the IH ip from its firewall.

7.3 Direct performance measurements

In order to prove the validity of the performance model created above, and to demonstrate the SDP performance results, each of the basic costs was measured using various scripts. Afterwards, using the equations created in 6.2 and the calculated basic factors, an estimation of the actual SDP performance was calculated for each action. This aproximation was later compared with the actual SDP performance in order to prove the validity of the performance model.

Every test and measurement was taken on a Acer-Aspire 5 laptop, the aspects of which can be seen on table 7.3.

Table 7.3: Acer-Aspire 5 A515-51-572Q

CPU	Intel Core i5-7200U 2.5GHz
RAM	8GB DDR4
Filesystem	256GB SSD

Each test was performed 100 times.

7.3.1 Measuring basic factors

Each of the previously declared basic factors, was tested and measured in order to find its execution time. The results are presented on table 7.5.

Table 7.4: Metrics Results

Metrics Results	
Factor	Value (sec)
UDP_{snd}	0.01
SSL_{con}	0.0023
SSL_{snd}	0.000108
CFG_r	0.0004
CFG_w	0.0003
AES_e	0.00039
AES_d	0.00001
$MD5_m$	0.000002
FW_{add}	0.001
FW_{rm}	0.0014

Notes on results:

- UDP_{snd} requires a total of 0.01s which seems like an extreme amount of time, considering it is a UDP packet. However SPA uses python's Scapy library to send the SPA packet , which adds an extra overhead to its packets [7].
- The size of the hashed/encrypted values provided as input to the MD5/AES functions while calculating its performance time, were equal to the fields used in the SPA itself.

7.3.2 SDP handshake measurements

Using the equations created in the previous section and the metrics, an approximation for the expected SDP authentication time can be calculated between two hosts. Therefore

$$SPA_{auth_{client}} = 0.000002 + 0.00001 = 0.000012s$$

$$SPA_{auth_{server}} = 0.0004 + 0.000002 + 0.00001 + 0.0003 + 0.001 = 0.002s$$

$$SPA_{auth} = 0.000012 + 0.01 + 0.002 = 0.012$$

$$SSL_{auth} = 0.0023 + 0.00039 + 0.000108 + 0.0004 + 0.00001 + 2 * (0.000108) = 0.003s$$

$$SDP_{auth} = 0.012 + 0.003s = 0.015s$$

The SDP authentication int the end-to-end measurements was calculated as equal to 0.01985s.

7.3.3 SDP data exchange measurements

Before measuring the SDP_{serv}, the service query performed by the IH in order to find an available AH must be calculated.

$$Q_t = 2 * (0.000108)s = 0.000216s$$

$$S_t = 0.000216s + 2 * (0.000108) + 0.0004 + 0.003 + 2 * (0.000108) = 0.004s$$

$$SDP_{init_{IH}} = 0.015 + 0.004 + 0.015 = 0.035$$

During the performance tests, the execution time of $SDP_{init_{IH}}$ was calculated as $0.036521ms$. Since the results satisfy the model, the next step is to proceed to SDP_{serv}.

$$SDP_{serv} = 3 * (0.000108) + S_{proc} = 0.0003 + S_{proc}$$

The test service used for the metrics, responds to message received via TCP. The cost of this service is minimal, so S_{proc} is considered to be equal to 0. The overall cost can be calculated as

$$SDP_{serv} = 0.0003s$$

The results gathered from end-to-end measuring on the process of sending packets to a service via an IH indicated that $SDP_{serv} = 0.000716s$.

7.3.4 Results

The final results are presented on table 7.5.

Table 7.5: Final Results

Action	Prediction(sec)	Result(sec)
SDP Authentication	0.015	0.0198
IH initialization	0.035	0.036521
SDP send	0.0003	0.000716

It is evident that the prediction model managed to predict correctly the outcomes of the SDP_{auth} and $SDP_{init_{IH}}$ procedures. However the prediction for SDP_{snd} is two times faster than the actual result. During the calculation of the SDP_{serv} prediction, the S_{proc} value was not taken into consideration (its value represented the performance cost of the actions done by the service during data exchange). In the performance tests, the service was a simple TCP server which responded to TCP packets. The overall cost of sending and receiving TCP packets was calculated to 0.000236sec. Therefore the deviation of the final result from the prediction was created by the cost of exchanging TCP packets with the service.

Chapter 8

Conclusion and Future Work

8.1 Future Ideas and implementations

The aim of this thesis was to create an SDP implementation, based on the specification document by CSA, as well as introduce new concepts and strategies to improve the system structure.

However the final implementation, has plenty of room for improvements and new ideas. Since the system was built with future implementations and extendability strongly into consideration, new features are fairly easy to implement and program using the existing mechanisms and structures.

8.1.1 Extended Controller API and new features

So far the SDP Controller API is very simple, it only consists of the start_server function which starts the controller and the add/edit_host, add/edit_AH functions which add new hosts into its configuration files.

However new API calls could be introduced to the controller, some ideas are the following

- get_AH_stats(aid) : returns various info for AH with the corresponding aid, number of clients, running time, client information, workload, number of running processes
- stop_host(aid) : terminates connection with specific host, IH or AH
- disable_ah(aid) : the CTL could use this function to disable temporalily some AHs when they are idle to save resources. Whenever an AH is disabled it transfers its state to the other active AHs and then it redistributes its clients to them to continue their operations
- enable_ah(aid) : activates AH when the network traffic increases

The final bullets adds a new important feature that currently is missing from the SDP implementation, IH-AH persistant state. Right now the connections are stateless, every time one of the hosts disconnects, it has to repeat the whole process again before connecting to an AH again. With some minor changes, the IH could connect to the AH again, and continue its previous actions.

A better algorithm for finding the appropriate AH for an IH_QUERY could also be created. The current solution to this is a simple Round Robin algorithm, each time a different AH is served. However choosing the new AH based on its number of clients and other factors could improve the implementation and distribute the performance load.

Finally, controller should be also able to connect and link AHs together so that requests could be redirected through more than one AH. This would proven to be an efficient mechanism as AH could be arranged into graphs or trees to distribute traffic through the network and not only act as end nodes.

8.1.2 Distributed sCTL structure

The current implementation can support only one sCTL. Such an architecture does not require a complicated file structure to store user data and information, so files are an appropriate solution for storage. However, an important addition to the SDP system would be to support more than one Controllers. A noSQL distributed database (such as MongoDB) could be used to exchange information between the controllers so that the system could scale easily.

The distributed controller structure could later be shaped in a tree-based structure, in order to support efficiently a greater number of clients. AHs could dynamically decide the controller to connect with and controllers could be able to exchange AHs between each other. Generally, a communication protocol for inter-sCTL communications is crucial to be developed, in order to create a structure that could be used in real-life applications and systems.

8.1.3 SDP Catalog Services

It was previously mentioned that clients in the SDP system should have a shared seed, for the communication encryption, beforehands. This creates a question, regarding the acquirement of the seed. SDP Catalog Services would be a dedicated server which handles such problems. Through it a client could register to the SDP application, in order to acquire its seed and aid and use them to connect with the SDP Controller. This web service could also have more uses such as:

- Inform the client of the controller closer to their location
- Serve information about specific controllers, such as IP, aid and make requests to get seed in order to connect with them
- Provide authentication, in the spirit of PKI, so that the client is sure that it is communicating with a valid SDP Controller

8.1.4 SDP Platform

As far as the current implementation is concerned, the client can connect and request access to any service on the SDP system. However, a future step would be to personalize this process, so that every client is linked only with specific services, and has its own profile inside the system, thus building the SDP platform (in the form of an application or a website). Through it, and by using the SDP Catalog Services, the user would be able to register new services and components to the system, see and communnicate with already existing services and customize its profile in a user friendly manner.

8.2 Epilogue

The primary aim of this thesis, was to implement/discuss the SDP and SPA protocols and mention their benefits. In general, both protocols provide an extra security layer for the services they protect, which means that an attacker would have to penetrate and exploit multiple security mechanisms in order to inflict damage to its target. The SPA protocol provides authentication prior to connection: clients can be authorized to a server without the need to establish a connection with it, therefore the server can block all its ports via the firewall and allow access to them only to authenticated clients. The SDP on the other hand, uses the SPA to authenticate its clients (IHs) and then provides them with gateways (AHs) to services hidden behind its structure.

As far as originality is concerned, SDP uses mechanisms that have been extensivelly used in the past: the AH strongly resembles a reverse-proxy, host-to-host communication uses protocols such as SSL/TLS, its architecture is similar to NAC and the firewall has been a basic tool in the security engineer's arsenal for years. However, the real breakthrough in SDP is not the SDP protocol per se, but the use of SPA as an authentication mechanism.

The SPA is a considerably new technology (proposed in 2005) which has not, yet, been extensivelly used in the computer field or the market. Similar to every new technology, a way to hack the SPA authentication system might be discovered in the future. However, due to the architecture of the SDP, even if SPA is hacked and exploited by attackers who will manage to find a way to penetrate the SPA firewall, the protected services will remain protected behind the Black Cloud. Whichever the damage is, it will be inflicted on the intermediate nodes who act as gateways to the services (AHs) and not to the services themselves. In that case, the engineers of the SDP system will soon discover the vulnerability, using the SDP's log files and will attempt to fix it. Even if a big amount of time is required to fix the issue, more AH nodes can be added to the system in order to handle the clients requests. Attackers would then have to rediscover the AH nodes and attack each one of them. This proccess (attack AH - create new AH) will be repeated as long as the attack lasts. Overall, the SDP seems to be successful in its primary goal, to protect its services, as even zero-day exploits (exploits that are used for the very first time) found in its most important security mechanism (the SPA), will not inflict damage upon them.

However, one can never be certain that a security mechanism will never be compromised. For example, a major security flaw was recently found on WPA2[2], a mechanism that, until now, was considered secure and replaced other insecure protocolls such as WPA. It is very possible that similar flaws will be discovered in the SDP system as well.

The primary aim of computer security is to provide defense against a multitude of attacks. However, in order to provide security against a threat, the threat has to be defined. The security engineers try to provide security against attacks that have already been used in the past. Security cannot be provided against attacks that have never been used before, only security measures can be taken, that once again have been designed based on previous experience. It is obvious that the attackers will always be one step ahead from the security engineers trying to stop them. The SDP creates security by creating multiple layers that secure its protected services. However each one of this layers might be broken in the future. This however does not deem this solution inefficient because a solution can never be considered efficient in general. As a matter of fact, the various tests executed during this thesis show that the SDP provides security against many popular computer attacks used today.

Security mechanisms exists because threats exist, and as long as threats exist and evolve so will the defense mechanisms built against them.

Appendices

Appendix A

SPA API

The library *spa* provides two modules. The first one is a function that sends SPA packets, while the second one is a server class, which configures the firewall, and sniffs the network for SPA packets(AH and sCTL use this class while IH uses the sending function). The SPA API is presented bellow

Class *sdpCtl.***spa_lib**spaListener(id,user_file, tls_p = 443, interface, block_all = True, no_seed_renewal = False, allowed_ips = []) Initializes a spaListener which sniffs for packets on *interface*. *id* is a unique identification, which will be added in the comments of every added rule in order to display its creator. *user_file* is the user file which contains client information (clients who are allowed to access the SDP system). *tls_p* is the port to allow connections on when a host is authenticated. *block_all* boolean, if set to true, adds the DROP all rule (for example AH does not add this firewall rule if it is in the same machine as the sCTL). When *no_seed_renewal* is set to true, AH does not change the seed of the client to the NEW_SEED value sent within the SPA packet. In case the *block_all* mode was set to true, *allowed_ips* is the list of IPs to be allowed on the network before blocking all connections (AHs add the IP of the controller).

- run()
  Starts sniffing packets. This function is non blocking. spaListener will run on a thread in the background.

- add_host(aid, password, seed)
  Adds a new host entry with these fields

- start_server()
  Start main execution thread

- is_alive()
  Checks if the spaListener is still running.

- terminate()
  Terminates the spaListener.

- remove_client(aid)
  Removes client identified by *aid* from the *user_file*.

- add_user_entry(aid, password, seed)
  Creates a new user entry with the input fields.

- edit_user_entry(aid, password = None, reset_rand = False, seed = None)
  Edits user identified by *aid.* In case *reset_rand* is set to true, it deletes previously used random numbers linked with this client.

- add_firewall_entry(ip, label)
  Add a new allow policy for *ip*, add *label* on the rule comments.

Function send_spa(aid, password, seed, new_seed, ip) Send an SPAp using the credentials given to address *ip*.

Appendix B

SDP API

The existing API for the SDP application is presented bellow.

Class *sdpCtl.***sdpCtl**(aid, ssl_p, server_crt, server_key) Initializes the controller with the corresponding aid, listens on port ssl_p for connections and uses the files server_crt and server_key to create ssl connections.

- add_AH(aid, services)
  Adds a new AH entry with these fields
- add_host(aid, password, seed)
  Adds a new host entry with these fields
- set_AH_services(self, aid, services)
  Sets the services for a specified AH
- start_server()
  Start main execution thread
- run()
  This function blocks execution until sCTL has been closed.
- terminate_ctl()
  Terminates all connections and threads of the controller.

Class *sdpHost.***IH**(aid, password, seed, ssl_port) Initializes an IH with the respective aid. The password is used for user authentication whereas the seed for host authentication. The ssl_port is the controller port to connect with.

- req_login()
  Logs in to the controller. Returns true if log in was successful, false otherwise.
- send_query(query_json)
  Sends an IH_QUERY. The query_json has been defined at 4.3.3. Returns True in case of success and false in the opposite scenario
- get_query_response()
  If the send_query function has been used, without requesting for a direct connection with an AH (by setting the service_id field), then the result of the query can be acquired by calling this function. The result is a json describing the available services or an empty dictionary in case of error.

- has_available_host(service_id)
  Returns true or false whether, the IH, knows an available AH to connect to for the provided service.

- get_available_host(service_id)
  Gets information of the AH host that provides the corresponding service if the query has been succesfull.

- connect_with_AH(server_crt, service_id)
  Connects with the AH for a service. Returns the MUX, if the AH approved the connection or null in the opposite case. Uses server_crt file to create ssl_connection and authenticate to AH.

- send_data(MUX, data)
  Sends data to channel created with AH with the corresponding MUX.

- get_data(MUX, timeout = QUEUE_CHECK)
  Gets data from channel created with AH with the corresponding MUX. This call is not blocking and will return null if no data is found in channel after a timeout (QUEUE_CHECK is equal to 2s)

- terminate_host()
  Terminates all the connections and thread listeners initiated during the execution of the IH.

Class *sdpHost.***AH**(aid, password, seed, host_crt, host_key, ctl_ssl_p, ah_ssl_p,) Initializes an AH with the respective aid. The password is used for user authentication whereas the seed for host authentication with the controller. The ctl_ssl_port is the controller port to connect with. The host_crt is for creating an ssl connection with the controller. The host_key is for creating an ssl server on port ah_ssl_p in order to listen for IH clients.

- req_login()
  Logs in to the controller. Returns true if log in was successful, false otherwise.

- run()
  This function blocks execution until AH has been closed.

- terminate_host()
  Terminates all the connections and thread listeners initiated during the execution of the IH.

Appendix C

SDP packets

C.1 LOGIN[1]

The first packet to be exchanged. Through the LOGIN packet, AH is authorized, and a new session is created by sCTL. E_AID is decrypted using the users' shared secret key and then compared to the AID field in order to authenticate user. The packet is displayed on table C.1.

Upon registration to the sCTL, the AH also creates its own firewall. Like the sCTL, it drops all incoming connections and packets except those coming from the Controller. Any client that wishes to communicate with the AH must authenticate itself via SPA.

Table C.1: LOGIN packet

Field	Value
OP	0x01
LENGTH	64
DATA	AID/E_AID

C.2 LOGIN_RESP

The response to LOGIN packet contains the identification number of the created session as well as the maximum time in msec from the last communication between the AH and the sCTL until the connection is closed (keepAlive value). The packet is displayed on table C.2

C.3 LOGOUT

This message is sent, either by the Controller to the AH or vice versa, in order to indicate the end of the communication. Whoever gets this message first assumes that the connection is over and terminates the session. The packet contains only the LOGOUT packet code : 0x03. The logout process can be seen on figure C.1

[1] The SDP specification document stated that the LOGIN packets should not include any other fields except from the packet header. However, in order to provide an additional layer of security, the AID and E_AID fields have been inserted.

Table C.2: LOGIN_RESP packet

Field	Value
OP	0x02
LENGTH	64
DATA	SESSIONID/KEEP_ALIVE

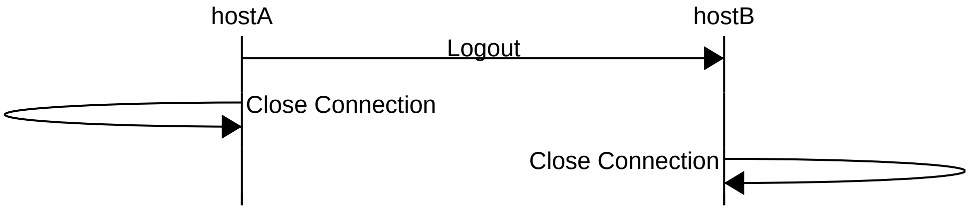

Figure C.1: Logout procedure

C.4 KEEP_ALIVE

Indicates that the client is alive. This packet is sent periodically by both members of the connection to keep the session open. In case of delay, the Controller or the AH terminates the connection (as seen on figure C.2). The packet contains only the KEEP_ALIVE packet code which is 0x04

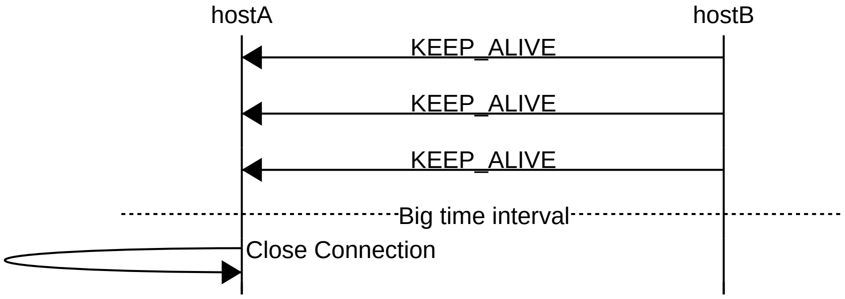

Figure C.2: Keep alive periodical send

C.5 AH_SERVICES

This packet is sent by sCTL to AH to inform it about the list of services it is going to provide. The list is formatted as a json array which contains the following fields:

- id : the id of the service

- name : the name of the service
- ip : the address of the service
- port : the port of the service
- type : the type of service (http, tcp, etc.). This instructs the AH how to connect with the service

When the list is received, the AH adds rules to the firewall allowing communication with these services. The packet structure is displayed on table C.3.

Table C.3: AH_SERVICES packet

Field	Value
OP	0x06
LENGTH	LEN
DATA	SERVICES_JSON

C.6 IH_AUTH

Sent by the sCTL to the AH when an IH user has been successfully authenticated and wants to use a service that this AH provides. The IH info is stored in a json object with the following fields:

- aid : the aid of the IH
- seed : the seed to use in the connection with the IH
- service_id : the id of the service requested

As soon as this message is received, the AH is ready to accept SPA packets from the IH. The packet is depicted on table C.4.

Table C.4: IH_AUTH packet

Field	Value
OP	0x07
LENGTH	LEN
DATA	IH_INFO_JSON

C.7 LOGIN

The first packet to be exchanged. Using a LOGIN packet, the IH is authorized and a new session is created by the Controller. The LOGIN packet has already been displayed on table C.1.

C.8 LOGIN_RESP

The response to LOGIN packet contains the identification number of the new session. It is shown on table C.5.

Table C.5: LOGIN_RESP packet(IH-sCTL)

Field	Value
OP	0x02
LENGTH	64
DATA	SESSION_ID

In contrast with the AH-Controller LOGIN_RESP, the KEEP_ALIVE field is not used here. sCTL terminates connection after an predefined period if the IH is inactive or when the IH has found an available AH to connect with.

C.9 LOGOUT

This message is sent by the Controller to the IH or vice versa in order to indicate the end of communication. Whoever gets this message assumes that the connection is over and terminates the session. This packet is exactly the same with the LOGOUT packet in AH-sCTL communication.

C.10 IH_QUERY[2]

When an IH has finally logged in, it can ask the controller to provide it with either a full set of services that it can use, or a subset thereof, by submitting a specific query. The query is a json object and its fields can be seen below:

- name : the name of the service
- type : the type of the service
- service_id : the ID of the service

When the IH is sure about the service it intends to use, it sends an IH_QUERY to the sCTL setting the service_id field with only the ID of the corresponding service. The sCTL then starts the procedure of connecting the IH with the designated AH. The packet is depicted on table C.6

C.11 IH_SERVICES[3]

After an IH_QUERY packet has been received, the IH_SERVICES packet is sent by the sCTL to the IH in order to inform the latter about the list of services matching the IH's query. The services list is a json object whose fields are described bellow

[2] This packet type did not exist in the SDP's specification.

Table C.6: IH_QUERY packet

Field	Value
OP	0x08
LENGTH	LEN
DATA	QUERY_JSON

- name : the name of the service
- type : the type of the service
- service_id : the ID of the service

The packet structure is shown on table C.7

Table C.7: IH_SERVICES packet

Field	Value
OP	0x06
LENGTH	LEN
DATA	SERVICES_JSON

C.12 AH_READY[4]

After receiving the IH_SERVICES packet with the service_id field set, the sCTL selects an appropriate AH to handle the request and informs it about the new IH through an IH_AUTH packet. When AH has acknowledged the AH_AUTH, the sCTL sends an AH_READY packet to the IH to inform it about its new gateway to the service. The packet includes information about the AH in json format, which is described below:

- ip : the IP address of the AH
- seed : the seed to be used during the communication
- service_id : the ID of the service that it is going to provide

The AH_READY packet format is depicted on table C.8

C.13 LOG_ACK

A client sends this to the server to acknowledge the receipt of the LOGIN_RESP packet. The packet is sent once the client has initialized a session. Once the server receives this message, it sets the clients session to active. The packet's code is 0x10.

[3] The IH_SERVICES message, did not exist in the original SDP protocol but was built into the implementation developed as part of this thesis.

[4] This packet was introduced in this version of SDP and was not included in the prototype

Table C.8: AH_READY packet

Field	Value
OP	0x09
LENGTH	LEN
DATA	AH_INFO_JSON

C.14 AHS_ACK

The AH sends this ACK to the sCTL in order to acknowledge the receipt of the AH_SERVICES packet. It is sent once the AH allows the new services to connect with it via its firewall. After receiving this packet, the sCTL can assign IHs to this AH for the services it provides. The packet's code is 0x11.

C.15 IHA_ACK

The AH sends this ACK to Controller in order to acknowledge the receipt of the IH_AUTH packet. It is only sent once the AH has updated its configuration files with client-related data. When this packet is received, the sCTL sends the AH_READY packet to the IH . The packet's code is 0x12.

C.16 OPEN_CONN_REQ[5]

Much like the LOGIN packet, the OPEN_CONN_REQ packet is used by the IH in order to ask AH to to connect with it (using the shared seed, exchanged on the service query step, in order to validate). The packet is displayed on table C.9. The AH opens a connection with the requested service during the communication. In case the communication fails, it sends an error message to the IH and terminates their connection.

Table C.9: OPEN_CONN_REQ packet

Field	Value
OP	0x07
LENGTH	64
DATA	AID/E_AID

C.17 OPEN_CONN_RESP

The OPEN_CONN_RESP packet is used as a response to the OPEN_CONN_REQ packet. It contains the MUX for this session. A MUX is a 32-byte string containing the service_id

[5] This packet follows the LOGIN packet structure and was introduced in this implementation

(first 16 bytes) and the session_id (last 16 bytes). In order to refer to this service, the IH and AH have to use this MUX. The packet structure is displayed on table C.10.

Table C.10: OPEN_CONN_RESP packet

Field	Value
OP	0x08
LENGTH	32
DATA	MUX

C.18 CONN_CLOSE

The CONN_CLOSE packet ends the connection between the IH and AH. The code of this packet is 0x0A.

C.19 DATA

The DATA packet is service data encapsulated inside an SDP packet, which is either forwarded, via the AH, to either the service or the IH. The packet is displayed on table C.11.

Table C.11: DATA packet

Field	Value
OP	0x09
LENGTH	LEN
DATA	SERVICE_PAYLOAD

Appendix D

Using SDP

The SDP and SPA libraries were implemented using Python 2.7.12. The current implementations can only run on machines using Linux.

In order to download the latest Python 2 and Python's package manager one has to execute the following commands:
Debian

```
$sudo apt install python python-pip
```

Red Hat

```
$sudo yum install python python-pip
```

Suse

```
$sudo zypper install python python-pip
```

After the python installation the user must install the required python libraries

```
pip install python-iptables pycrypto bs4 uuid scapy\
        netifaces pyyaml crypto
```

The APIs for SDP and SPA are now ready to use.

Examples of SPA and SDP use have been provided on the following files:

- sdp_proto/startCTL.py : Starts the SDP Controller
- sdp_proto/startAH.py : Starts an SDP AH and connects it with the controller
- sdp_proto/startAH.py : Starts an SDP AH and connects it with the controller
- sdp_proto/ping_client.py : The ping application described in the 4th and 6th chapters
- sdp_proto/pong_service.py : The pong service described in the 4th and 6th chapters
- sdp_proto/ih_proxy.py : The IH proxy built on chapter 6
- sdp_proto/ah_proxy.py : The proxy service built on chapter 6

Bibliography

[1] Iot calamity: the panda monium, Feb 2017. Available at `http://www.verizonenterprise.com/resources/reports/rp_data-breach-digest-2017-sneak-peek_xg_en.pdf`.

[2] KRACK Attacks: Breaking WPA2, Dec 2017. Available at : `https://www.krackattacks.com/`.

[3] codingplanets/Overload-DoS, Feb 2018. Available at `https://github.com/codingplanets/Overload-DoS`.

[4] Goodbye, NAC. Hello, software-defined perimeter, Feb 2018. Available at `https://www.csoonline.com/article/3141930/security/goodbye-nac-hello-software-defined-perimeter-sdp.html`.

[5] netstat(8) - linux man page, Feb 2018. Available at : `https://linux.die.net/man/8/netstat`.

[6] Software Defined Perimeter - Cloud Security Alliance, Jan 2018. Available at `https://cloudsecurityalliance.org/group/software-defined-perimeter/#_overview`.

[7] Speeding up scapy packets sending » To Linux and beyond !, Feb 2018. Available at `https://home.regit.org/2014/04/speeding-up-scapy-packets-sending/`.

[8] Katherine Albrecht and Liz Mcintyre. Privacy nightmare: When baby monitors go bad [opinion]. *IEEE Technology and Society Magazine*, 34(3):14–19, 2015.

[9] Sabeel Ansari, SG Rajeev, and HS Chandrashekar. Packet sniffing: a brief introduction. *IEEE potentials*, 21(5):17–19, 2002.

[10] Cody Brocious. My arduino can beat up your hotel room lock. *Black Hat USA 2012*, 2012.

[11] Huseyin Cavusoglu, Birendra Mishra, and Srinivasan Raghunathan. The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers. *International Journal of Electronic Commerce*, 9(1):70–104, 2004.

[12] Marco De Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O de Vivo. A review of port scanning techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[13] Yvo Desmedt. Man-in-the-middle attack. In *Encyclopedia of cryptography and security*, pages 759–759. Springer, 2011.

[14] Toby Ehrenkranz and Jun Li. On the state of ip spoofing defense. *ACM Transactions on Internet Technology (TOIT)*, 9(2):6, 2009.

[15] Diaa Salama Abd Elminaam, Hatem Mohamed Abdual-Kader, and Mohiy Mohamed Hadhoud. Evaluating the performance of symmetric encryption algorithms. *IJ Network Security*, 10(3):216–222, 2010.

[16] Shahzadi Farah, Younas Javed, Azra Shamim, and Tabassam Nawaz. An experimental study on performance evaluation of asymmetric encryption algorithms. In *Recent Advances in Information Science, Proceeding of the 3rd European Conf. of Computer Science,(EECS-12)*, pages 121–124, 2012.

[17] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[18] Russell Housley, Warwick Ford, William Polk, and David Solo. Internet x. 509 public key infrastructure certificate and crl profile. Technical report, 1998.

[19] Martin Krzywinski. Port knocking. *URL: http://www. linuxjournal. com/article/6811*, 2003.

[20] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[21] Michael R Lyu and Lorrien KY Lau. Firewall security: Policies, testing and performance evaluation. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, pages 116–121. IEEE, 2000.

[22] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. Internet denial of service: Attack and defense mechanisms (radia perlman computer networking and security). 2004.

[23] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal*, 8(5), 2017.

[24] Bart Preneel. Cryptographic hash functions. *Transactions on Emerging Telecommunications Technologies*, 5(4):431–448, 1994.

[25] Gregor N Purdy. *Linux iptables Pocket Reference: Firewalls, NAT & Accounting.* " O'Reilly Media, Inc.", 2004.

[26] Loki Radoslav. Ssh file transfer protocol. 2012.

[27] Michael Rash. Single packet authorization with fwknop. *login: The USENIX Magazine*, 31(1):63–69, 2006.

[28] Michael Rash. Single packet authorization. *The Linux Journal (April (156))*, 2007.

[29] Eric Rescorla. *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading, 2001.

[30] Eric Rescorla and A Schiffman. The secure hypertext transfer protocol. 1999.

[31] Howard A Seid and ALbert Lespagnol. Virtual private network, June 16 1998. US Patent 5,768,271.

[32] Gustavus J Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, 1979.

[33] James E Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[34] Paul Syverson. A taxonomy of replay attacks [cryptographic protocols]. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 187–191. IEEE, 1994.

[35] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) connection protocol. 2006.