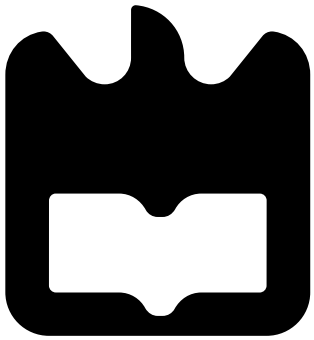
Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2016

João Filipe Teixeira Simões

# Novos Paradigmas de Controlo de Acesso a Máquinas na Internet

# New Paradigms for Access Control on Internet Hosts



Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2016

João Filipe Teixeira Simões

# Novos Paradigmas de Controlo de Acesso a Máquinas na Internet

# New Paradigms for Access Control on Internet Hosts

*"I never dream of success. I work for it."*

— Estee Lauder

**João Filipe Teixeira Simões**

# Novos Paradigmas de Controlo de Acesso a Máquinas na Internet

# New Paradigms for Access Control on Internet Hosts

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor André Zúquete, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Professor Doutor Paulo Salvador, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

**o júri / the jury**

presidente / president

**Professor Doutor Rui Luís Andrade Aguiar**
Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

**Professor Doutor Pedro Miguel Alves Brandão**
Professor Auxiliar da Faculdade de Ciências da Universidade do Porto (Arguente)

**Professor Doutor André Ventura da Cruz Marnoto Zúquete**
Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (Orientador)

**agradecimentos / acknowledgements**

A realização desta dissertação de mestrado marca o fim de uma importante etapa da minha vida. É com todo o gosto que aproveito esta oportunidade para agradecer a todos aqueles que de alguma forma contribuiram durante este longo ciclo.

Em primeiro lugar, destaco a importância da minha família. Agradeço aos meus pais, Carlos Jorge e Maria Manuel, por serem modelos de trabalho, mostrando constante apoio, preocupação e ânimo. Sem eles este enorme passo não seria de todo possível. Aos meus irmãos, Nuno, Tomás e Gabriel, que nos momentos de maior cansaço conseguiam proporcionar-me um genuíno e ilógico repouso. À Mariana por ter sido um incessante ponto de suporte, conforto e alegria.

Reconheço um especial agradecimento ao meu orientador, André Zúquete, que me possibilitou o culminar desta etapa através dos seus conhecimentos, da sua confiança, da sua boa disposição, e da sua permanente exigência para mais e melhor. Também ao meu co-orientador, Paulo Salvador, pela sua sabedoria, transparência, e aptidão para o esclarecimento dos temas adjacentes.

Por último, agradeço aos amigos pelo companheirismo, força, amizade e confiança que depositaram em mim, e que contribuiram com muitos bons momentos que estão guardados para o resto da vida. A todos os que me ajudaram a ser quem sou, resta-me apenas não desiludir. Muito obrigado.

**Palavras-chave** Controlo de Accesso na Rede, Segurança de Serviços, Módulo da Firewall, HMAC, Túnel Seguro

**Resumo** Os serviços de rede fazem uso da Internet para trocar informação com clientes que os solicitam. Esta informação segue, naturalmente, uma rota de redes inseguras e desconhecidas. De tal modo, não existe uma certeza absoluta que o tráfego que flui entre os clientes e os servidores é autêntico e é de facto originário de entidades conhecidas e legítimas. Também não existem políticas claramente definidas ao nível da rede, que autorizem utilizadores, ao invés de máquinas, a acederem a serviços remotos.

De maneira a mitigar o acesso não autorizado a serviços de rede, duas aproximações são frequentemente adotadas. A primeira aproximação conta com a inserção de *firewalls* para proteger o fornecedor de serviços. No entanto, a informação usada para fazer controlo de acesso é baseada nas camadas intermédias da pilha de protocolos de rede. Isto possibilita às *firewalls* controlar o acesso tendo em conta os sistemas de origem, mas não os seus utilizadores. Por outro lado, a segunda aproximação apresenta o conceito de controlo de acesso baseado em utilizadores. Contudo, este mecanismo de segurança é apenas aplicado nas camadas mais altas da pilha de protocolos, através de aplicações complexas e totalmente inconscientes de problemas de segurança ao nível do IP.

O sistema proposto combina o melhor dos dois mundos ao permitir que a autenticação e autorização de utilizadores sejam feitas ao nível da rede. A solução implementa um novo módulo da *firewall* ao nível do *kernel* para validar ligações estabelecidas, através de configurações trocadas previamente num canal seguro. Aceder a serviços remotos torna-se um processo devidamante controlado onde os utilizadores são reconhecidos como legítimos no lado do servidor.

**Keywords** Network-Level Access Control, Service Security, Firewall Module, HMAC, Secure Tunnel

**Abstract** Network services make use of the Internet to exchange information with requesting clients. This information follows a path of naturally unsecured and unknown networks. As such, there is no certainty that traffic flowing between clients and service providers is authentic and is actually originated on known and legitimate entities. Also there are no clearly defined network-level policies that authorize users, instead of hosts, to access remote services.

In order to mitigate the unauthorized access to network services, two conceptual approaches are usually adopted. The first relies on the deployment of firewalls protecting the service providers. However, information used to perform access control is based on intermediate layers of the network protocol stack. This enables firewalls to control the access based on originating physical hosts, but not on actual users. On the other hand, the second approach presents the concept of access control based on users. This security mechanism however, is only applied too far up the protocol stack, through heavyweight applications that are completely unaware of IP security issues.

The proposed system combines the best of both worlds by enabling authentication and authorization of users at the network level. The solution implements a new kernel-level firewall matching module to validate incoming connections, according to configurations previously exchanged through a secure tunnel. Accessing remote services becomes a duly controlled process where accessing users are confirmed as legitimate on the server side.

# Contents

# Acronyms

| | |
|---|---|
| **ACL** | Access Control List |
| **AH** | Authentication Header |
| **API** | Application Programming Interface |
| **CIFS** | Common Internet File System |
| **DAC** | Discretionary Access Control |
| **DCE** | Distributed Computing Environment |
| **DDoS** | Distributed Denial of Service |
| **DHCP** | Dynamic Host Configuration Protocol |
| **DNS** | Domain Name System |
| **DOI** | Domain Of Interpretation |
| **DoS** | Denial of Service |
| **DRBAC** | Distributed Role-Based Access Control |
| **ESP** | Encapsulating Security Payload |
| **FTP** | File Transfer Protocol |
| **GUI** | Graphical User Interface |
| **HMAC** | Keyed-Hash Message Authentication Code |
| **HTTP** | Hypertext Transfer Protocol |
| **IANA** | Internet Assigned Numbers Authority |
| **IBC** | Identity-Based Cryptography |
| **IDS** | Intrusion Detection System |
| **IP** | Internet Protocol |
| **IPS** | Intrusion Prevention System |
| **IPsec** | Internet Protocol Security |
| **IPv4** | Internet Protocol Version 4 |
| **IPv6** | Internet Protocol Version 6 |
| **LDAP** | Lightweight Directory Access Protocol |
| **LRBAC** | Location-aware Role-Based Access Control |
| **LTS** | Long-Term Support |
| **MAC** | Mandatory Access Control |
| **MAC** | Message Authentication Code |

**MD5** Message Digest Algorithm 5
**MTU** Maximum Transmission Unit

**NAPT** Network Address and Port Translation
**NAT** Network Address Translation
**NFS** Network File System
**NIST** National Institute of Standards and Technology
**NUAC** Network-level User Access Control

**OKTCP** Option-Keyed Transmission Control Protocol
**OSI** Open Systems Interconnection

**PAT** Port Address Translation
**PIN** Personal Identification Number
**PPTP** Point-to-Point Tunneling Protocol

**QR** Quick Response code

**RADIUS** Remote Authentication Dial In User Service
**RAM** Random Access Memory
**RBAC** Role-Based Access Control
**RFC** Request For Comments
**RFID** Radio-Frequency Identification
**RIP** Routing Information Protocol

**SCP** Secure Copy Protocol
**SCRAM** Salted Challenge Response Authentication Mechanism
**SFTP** Secure File Transfer Protocol
**SHA1** Secure Hash Algorithm 1
**SIP** Session Initiation Protocol
**SNMP** Simple Network Management Protocol
**SOCKS** Socket Secure
**SSH** Secure Shell
**SSL** Secure Sockets Layer
**SSTCP** Spread Spectrum Transmission Control Protocol

**TCP** Transmission Control Protocol
**TGTCP** Tailgate Transmission Control Protocol
**TLS** Transport Layer Security
**TRBAC** Temporal Role-Based Access Control

**UDP** User Datagram Protocol

**VFS** Virtual File System
**VoIP** Voice over Internet Protocol
**VPN** Virtual Private Network

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

## 1.1 Motivation

In the digital universe, worldwide communication is achieved through the usage of the Internet, the well-known global system of interconnected networks. This communication is generally represented as information exchanged between entities that are physically far apart. Most of the times, the information needs to be provided to a requesting user. A *client* requests access to information which is to be delivered by a service provider in the network, commonly known as a *server*.

With the growing number of *cyber-attacks*[1], network services remain a primary target for attackers, since they are deployed over the Internet and are visible to everyone. Port scanning techniques [15] allow for an attacker to discover entrances to a server. These entrances come in the form of open ports that may be exploited by attackers. Considering that there is a database with static mappings between port numbers and services[2], attackers are more easily able to disturb the access of legitimate users to specific services. Keeping in mind that such attacks come from unknown networks, controlling service access is therefore an increasingly eminent security concern for service providers.

A solution that has been adopted to mitigate the unauthorized access to services is based on the deployment of firewalls around the target systems. Yet, the information used to perform access control is based on the network and transport layers of the OSI Reference model. This means that any user is given access to the system solely based on the network or transport information. This information can be easily forged by an attacker trying to impersonate an authentic and well-intended user.

Firewall rules that use the information of the network and transport layers also become too difficult to manage for mobile agents. For mobile servers, it rules out the possibility to define different access policies and client availability on different networks. For mobile clients, it becomes impractical to associate network addresses to real clients in a dynamic way.

Controlling access at such low level without additional information is therefore an evident adversity, since it is simply based on IP addresses and service ports. This information does not allow for an unequivocal association between a user's identity and its host address, specially if NAT is deployed in-between the communication endpoints. Despite dealing with a part of the problem, the usage of a VPN essentially allows authorized users to connect to a server, it

[1] `http://www.digitalattackmap.com/`

[2] `http://www.iana.org/assignments/port-numbers`

does not allow for a simple filtering of users based on the services they are authorized.

As such, there is a demand for a high-level, computationally cheap and uncomplicated security system, that permits the access control based not only on network-level information, but also on supplementary knowledge of user identities' profiles. The system should also enable the early discard of unauthorized attempts and conceal itself to attackers, while providing transparent service access to authorized users. For all purposes, the system has to be understandable by network security administrators that have no need to understand concepts of protocols, addresses, layering, and so forth, at a deeper level. The system's administrator should only need to specify user permissions to given services, in order to make the system function properly and transparently.

## 1.2 Contribution

Application-level security deals with authentication and authorization of users in an application-only context, disregarding network information. A given user's identity is exclusively verified at a higher level on the TCP/IP stack, which is rather late for a penetrating attack onto the physical system. On the other hand, firewalls tend to implement security at the network and transport layers. They cannot, however, define the concept of users and access permissions merely based on information present on such layers. The proposed system combines the best of both paradigms to contribute with a solution where the concept of authentication and authorization of users is enabled at the network level.

Practically, the main purpose of the new system is to provide a **trustworthy and controlled access to remote network services**. This is accomplished bearing in mind several premises that translate into objectives:

1. Administer end-to-end protection between the client and the server, regarding message forgery;
2. Control the access onto the entire port range of the server machine;
3. Support the most widely used types of traffic;
4. Integrate, and not replace, into existing protocols of the TCP/IP stack;
5. Work with private networks as well as with the public Internet.

Concerning the security present on the interconnected public networks, it is assumed by the proposed system that an attacker can **eavesdrop**, **forge** and **inject** traffic on any direction of the information flow between the client and the server.

The new system is essentially implemented as a mechanism, parallel with a firewall, that decides on the eligibility of a given request received at the server-side. A firewall is therefore enriched with a module that knows how to filter traffic of different originating users, based on configurations set by the server. In order to be authenticated and given access to a specific service, a client must necessarily supply his credentials and then transmit traffic containing information associated with its identity. This information is represented as *non-obvious* data that stands unintelligible to eavesdroppers.

More specifically, it is possible to establish three major stages for securely accessing the provided network services. The first step is to authenticate the client onto the remote server. Upon a successful authentication, there is a process for data exchange where messages are

authenticated and subsequently verified at the server side. The final stage comprises the client's authorization based on both the initial authentication's credentials and the exchanged messages' appendices.

The initial client authentication is granted through a well-known secure channel. The authentication is successful if the provided identity matches a previously defined user profile on the server. Along with the data exchange, the traffic suffers a transformation before exiting the client's machine, and is appended with security information. This information is an access token that only authorized clients possess. The access control is executed at server-side with access policies created beforehand, according to different environments. The users are authorized the access to provided services, based on permissions that reflect the designated administrative policies.

Given this, the system shapes a level of abstraction due to the formation of important and concise properties. As such, it enables:

- Transparency to running applications;
- Inconspicuousness of exchanged traffic, as it *appears* to be natural;
- Independence of running protocols;
- Indifference on local or public networks, thus free of NAT obstacles.

Considering such properties, it should be noted that the system does not intend to replace applicational security, it purely supplements it with additional security mechanisms and user concepts at a lower level. Also it stands as a proof of concept solution, therefore it is applied only to version 4 of the Internet protocol. IPv6 however, is taken into consideration for just about the same aspects. This system is therefore considered wide and generic for current TCP/IP stack implementations.

## 1.3 Organization

This dissertation is essentially organized in seven chapters, where the first corresponds to this introduction and the last substantially describes ending conclusions and future work.

A theoretical context is presented in Chapter 2, covering fundamental concepts needed to effectively understand the mechanisms and structures implemented by the system. The context is mainly focused on networking and security concepts, while also briefly describing internal system communication, precisely on user to kernel space. It should be given emphasis to Next-Generation Firewalls since they describe exactly what the system intends to contribute. Most of the examined concepts are intimately associated with the impending system, while some are strictly referenced to provide a notion of concurrent available solutions, that can be compared with the ones implemented.

Chapter 3 encompasses several worth mentioning solutions, closely related to each of the system's cornerstones, specifically the concealment, authentication and authorization. The TCP-layer name service stands as the starting point for the system's hiding, while more complex solutions detailed by lightweight concealment techniques are taken into further substantial consideration. Significant contributions of role-based authorization must also be highlighted for withstanding the system's major objectives on access control.

In Chapter 4, the architecture of the proposed solution is carefully described. The description of the architecture follows the natural flow of information on the system, so that

not only it may be easily understandable through well-defined steps, but the transition to the implementation may be as intuitive as possible. The architecture also defines the conception of a new protocol used on the traffic flowing within the system. In general, the architecture relates the decisions made into distinct operational elements.

The actual implementation of the system is outlined in Chapter 5. There is a translation of the architecture's abstract constituents into concrete components that communicate between each other. The implementation's description results in: the specification of structures used to correctly fulfill protocol requirements and communication mediums, and the development of the specific components deployed on the system.

For the purpose of evaluating the implemented system, functional and performance tests were executed. In chapter 6, these tests are thoroughly detailed of their procedure, expected and obtained results. The results are then analyzed and compared to other transmission types to ultimately assess the system's behavior and efficiency.

The dissertation's conclusions summarize the key points of this work, while also referring some drawbacks of the devised architecture. The future work addresses the identified handicaps of the system and intends to complement with solutions that may be studied for future development. Both these matters are considered and reviewed in Chapter 7.

Before the end of the dissertation, an appendices section is provided to further delve into the implementation of the system, not only on the developed source code, but also on the environment used for its deployment.

# Chapter 2

# Theoretical Context

Within this chapter, a context of theoretical concepts is provided in order to fully grasp the broad extent of the proposed system. The concepts contemplate a background that covers three preeminent areas: networking, security, and kernel-user communication. These areas are closely associated to the proposed system's objectives, as well as with its architectural constituents. Some of the topics are merged together over different areas.

Regarding networking, a significant depiction of network communication and the system's used protocols is shown. Also, the firewall and port scanning topics are addressed for their importance on the overall specified goals. NAT is still discussed as a potential component present on most real networks.

In relation to security, the following sections attend mechanisms of secure channels and cryptographic authentication methods. These topics are relevant for the decision making on the security aspects of the system.

The access control is widely referenced to assert the different possibilities of user authentication and authorization on the system. A brief explanation of some of the existing virtual filesystems is also supplied, with emphasis to the *Configfs* filesystem, the one introduced in the system.

## 2.1 Network Communication

Traditional communication can be simply defined as the exchange of information between two given entities. As such, network communication inherits that definition with some nuances. The information exchanged can be identified as data while the entities are generalized into endpoints of the communication. In order to provide a correct interpretation of the information exchanged, some rules of communication must be applied. Following the same analogy, the set of rules can be translated into a **protocol** that specifies the correct way of exchanging data. The most common way of transmitting data along the network is by sending the message, split into relatively small chunks called **packets**, thereby describing a packet-switched network.

A packet is a formatted unit of data containing a group of bits or bytes, sent on a packet-switched network. Specifically, the contents of a packet can be divided into two kinds of data: control information and actual data. The first part of the packet, usually known as **packet headers and trailers**, provides the information needed to control the delivery of the actual **payload** data through the network. Source and destination addresses are an example

of control information.

The data delivery is ensured by the control information present in the packet, while the protocol guarantees that the endpoints are following the same set of rules and methods to ensure the success of such delivery. Protocols are divided into categories representing their main function. They can be inserted into the transport layer, in both TCP/IP model and OSI model, since their primary task is to provide **end-to-end communication**. This type of communication is generally categorized as either connection-oriented, implemented in the Transmission Control Protocol (TCP) or connectionless, implemented in User Datagram Protocol (UDP). This layer also establishes the concept of port that provides process-specific transmission channels for network services. Every port number has its own instantiation in both UDP and TCP.

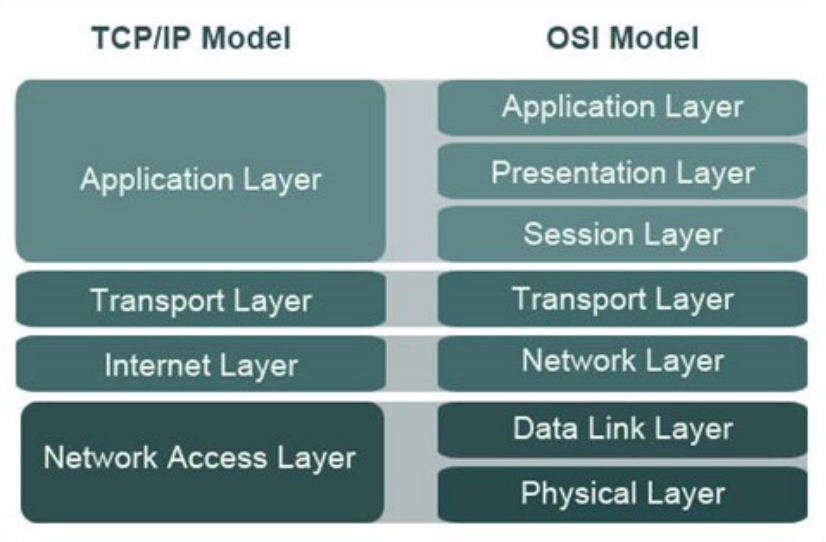



Figure 2.1: TCP/IP model vs. OSI model. The OSI model provides a greater granularity of layers comparing to the TCP/IP model.

Not only there are protocols that implement end-to-end communication at the transport layer, but there are also some carrier protocols that are categorized as part of the Internet layer. Such protocols carry information considering different conceptual approaches. The Internet Control Message Protocol (ICMP) and the Generic Routing Encapsulation (GRE) are protocols that, despite being part of the Internet layer, are also transmitted using the Internet Protocol for information delivery.

### 2.1.1 UDP

User Datagram Protocol, or UDP, is one of the most important transport-layer protocols used worldwide. Its rating is due mainly to its conceptual and practical simplicity, lightness and fastness, and variety in applicational uses. This protocol is sometimes also referred to as UDP/IP because it uses the Internet Protocol as the underlying protocol. UDP is a message-oriented protocol, therefore the messages correspond to datagrams that are sent in the form of packets. RFC 768 [44] determines the format of the packets used by UDP.

### UDP Datagram

Like most packets, UDP packets consist of control information and data octets. UDP's control information is encapsulated into the UDP header (see Fig. 2.2) which comprises 4 fields, each of which is 2 bytes long, resulting in a minimum packet size of 8 bytes. The data section may have a variable size.

The source port number field identifies the sender's port, and is usually assumed to be the destination port on forthcoming replies. When used, this port specifies the originating endpoint connection used by some application on the source host. Note that this field is optional and therefore when not used should be set to zero. The destination port follows an analogous principle, it identifies the endpoint connection on the destination host. In contrast to the source port, the destination port number is required, so that the packet can be successfully delivered to network services or applications running on the destination host.

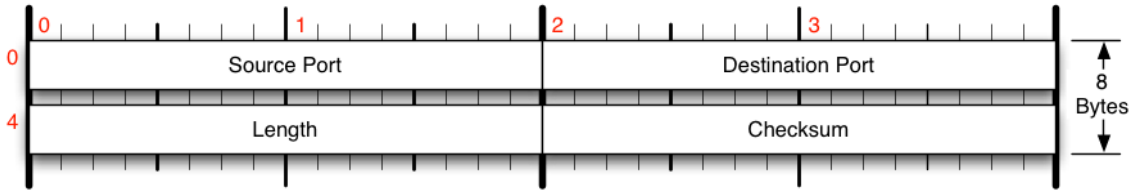



Figure 2.2: UDP header. The source port is an optional field while the destination port is required for service distribution.

The length field maintains the total size of the entire datagram, meaning both UDP header and payload. The minimum length is 8 bytes when there is no data on the payload and this field only includes the size of the header. The Internet Protocol specification [46] recommends hosts to be prepared to accept datagrams of up to 576 octets to allow a reasonable data block size of 512 octets while the rest can be used for headers for the different underlying protocols.

Error-detection during transmission can be achieved through the checksum algorithm. The computation method consists of simply summing up 16-bit words using one's complement, bearing in mind a *pseudo header* format comprised of fields from the IP header, the UDP header and the data. The sum is then one's complemented to yield the value for the checksum field. Despite being an optional field on IPv4, it is mandatory on IPv6 and respects a slightly different computation according to the IPv6 header [16].

### UDP Features

Being a minimalist and simple protocol, based on datagram messages, UDP is classified as a connectionless protocol that does not set up a dedicated end-to-end connection for exchanging information. The information is sent from the source to the destination host without taking into account the readiness of the receiver. Specific features, that contrast with protocols present on the same layer, can characterize UDP for its way of operation:

- Unreliability. Being the main characteristic of UDP, the message is not guaranteed to arrive at the destination endpoint. The lack of control mechanisms such as retransmission, datagram ordering, connection tracking and state maintenance, reflects the unreliability of the protocol, allowing for less overhead during transmission;

- Datagrams as messages. A rapid transmission is assured due to the lightweight headers, as well as message reading as a whole;
- Half-duplex transmission. There is no clear definite state of host-to-host connection so the communication is generally considered unidirectional;
- Integrity verification via checksum. Transmission errors are detected in the entirety of the UDP datagram;
- Port numbers. Permits applicational multiplexing on port-binded sockets, either on the sending or receiving host.
- No congestion control. There is no control whatsoever of the sent packets entering the network due to the inherited unreliability.

All the above features concede simplicity, lightness and speed to UDP, making it particularly useful for: DNS services and protocols such as SNMP, RIP and DHCP, either for its query-like, short response transactions or unidirectional availability for broadcast and multicast; and services that use real-time streaming, VoIP and online games, where loss of packets inherent to UDP's unreliability is not usually a fatal issue. Solutions for reliability may be added at the application level, issuing for developers to implement them on the applications.

### 2.1.2 TCP

A complementary approach to UDP is the Transmission Control Protocol [47], a connection-oriented protocol that grants end-to-end reliable inter-process communication. This protocol provides an abstraction of the communicational service between applications and the underlying network connection. Its connection-oriented designation brings the notion of *data stream* communication. This method of communication is applied to both end-hosts allowing for a bidirectional information exchange, where messages are delivered sequentially and split into smaller units of data for each direction. The small pieces of data used in TCP are often labeled as TCP segments which are enclosed and sent inside IP datagrams.

#### TCP Segment

Processes of applications are responsible for providing data streams onto the TCP. The TCP packages the data into segments and calls the Internet Protocol to arrange the delivery of the encapsulated segments to the correct destination host. Each TCP segment is comprehended by a total of 10 required fields and 1 optional extensible field, representing the header (see Fig. 2.3), and the data section containing the payload data descendant from the applications.

The source and destination ports are 16-bit fields that introduce the TCP segment header. Both these fields map to endpoint sockets used by the originating host and destination host applications, respectively. Like UDP, their main functionality is to provide correct delivery of information to different applications running on a target host.

The sequence number and the acknowledgment number are two very important fields that implement the basis of this protocol's communication reliability. This is a direct consequence of the decision of numbering the payload data at the octet level. The sequence number corresponds to the first data octet in the segment accumulated with the number of previously

sent octets. The acknowledgment number specifies the next sequence number the sender is expecting to receive while also acknowledging the receipt of all prior octets. During the connection establishment, acknowledges are sent to accept the other end's initial sequence number.

The data offset field indicates where the data starts inside the segment, or another possible interpretation, the size of the TCP header, in 32-bit words. The minimum header size is 5 words (20 bytes), corresponding to a segment with no options, and the maximum is 15 words, allowing for up to 40 bytes of options.

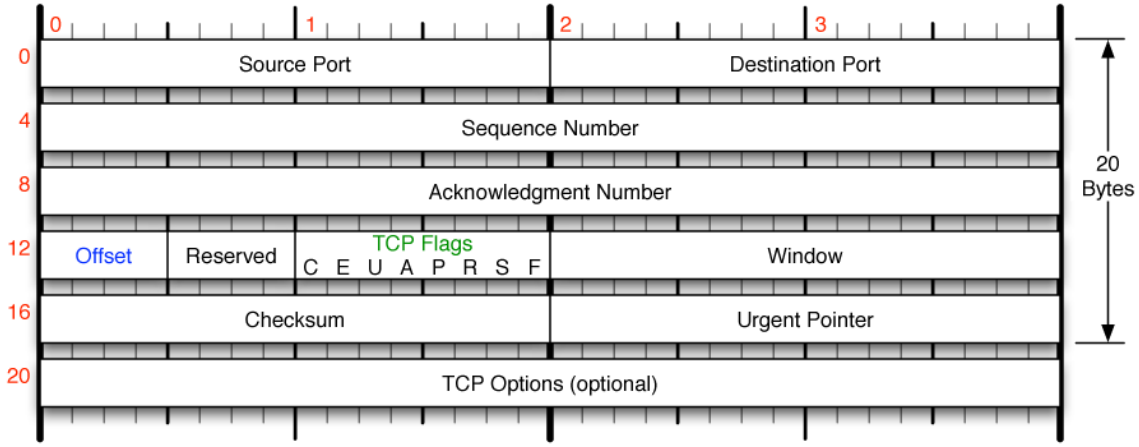



Figure 2.3: TCP header. TCP flags are represented by their initial letters. The offset field identifies the number of 32-bit words that precede the payload. The payload follows TCP options (if present), or the urgent pointer otherwise.

The following set of fields, often called as flags, control TCP's operation. Initially the protocol had 6 flag bits (URG, ACK, PSH, RST, SYN, FIN) but latter two more bits were introduced as flags [48] (CWR, ECE). The last 2 bits deal with issues related to network congestion notification.

The window field gets its name from the amount of data octets, including the one referenced by the acknowledgment number, the sender of the segment is willing to receive in his buffer. The receive window grants the possibility of flow control, one of TCP's features.

TCP's checksum follows the same computational method of UDP with a similar *pseudo header* format including some fields from the IPv4 header, the TCP header and the payload data. When the communication is made through IPv6, the elements used as part of the pseudo header change slightly and relate to the IPv6 header format accordingly.

Sometimes the data one intends to transmit is considered as urgent, for that the urgent flag is activated and the urgent pointer field indicates the next sequence number following the urgent data. It is to be noted that the activated urgent flag together with the urgent pointer does not, whatsoever, translate into any additional process on the network itself. It may, however, alter the data processing on the remote host.

Options, if present, occupy the last space of the header. A maximum of 40 bytes may be used however, this does not reflect an exact maximum amount of options present in a TCP segment. This is true due to two facts: options may begin on any octet boundary and may have different sizes. This variety in option length results in the possibility of the list of options

ending in a non 32-bit boundary thus having TCP the need to zero-pad the remaining header to reach the data offset value.

#### TCP Features

Although a complex protocol, TCP provides many important features for interactions that need increased reliability and security. Mainly, this is achieved by having the core of the protocol based on control over the connection. TCP is a connection-oriented protocol, meaning that it requires previous establishment of a communication channel dedicated to data transferring through streaming between two hosts connected by the Internet. Due to its definition on the operational methodology, this protocol emphasizes on accurate delivery in contrast to timely delivery. TCP is therefore characterized by the following features:

- Reliability. Each segment is assured to be received at the opposite endpoint. The concepts of sequence and acknowledgment numbers, combined with state preservation, answer packet duplication and disorder, while retransmission offers solution for packet damage and loss;
- Segments as parts of the message. Despite the slowness due to the increased overhead on TCP headers, data can be streamed and buffered while waiting to be delivered to the application;
- Full-duplex transmission. A bidirectional information exchange is made possible through the three-way handshake on connection establishment;
- Integrity control. Errors during transmission can be detected by a checksum algorithm;
- Port numbers. Multiplex different applications, permitting several end-to-end connections to be established on different port pairs.

Adding to the previous list, TCP also makes use and implements several mechanisms that support two very important features:

- Flow control. The sending host knows exactly how many more bytes it can send to the receiving host. This is possible through window advertisements according to the latter's buffer capacity and processing power;
- Congestion control. It uses a number of mechanisms to control the rate of data entering the network [3], opposed to flow control that supervises rate of data entering the receiving endpoint. Congestion avoidance is the basis to achieve high network performance [11].

TCP's reliability is very useful for many upper layer protocols and applications such as HTTP, SSH, file transferring protocols like FTP, and e-mail delivery. In these cases, data must be perfectly delivered without any packet loss.

### 2.1.3 ICMP

The Internet Control Message Protocol, as the name implies, is a protocol for controlling Internet communication. The control is performed based on diagnostics and reports generated

in response to errors occurred during IP packet processing. The reported issues are then addressed back to the originating entity for further evaluation. ICMP is considered to be an integral part of IP, for being a special case during the latter's processing. ICMP messages are encapsulated into the IP protocol, exhibiting the number 1 on the IP header's protocol field, and represent the vehicles for relaying such control information.

### ICMP Datagram

ICMP messages follow the ordinary packet structure containing an header section and a payload section. The 8-byte header, illustrated in Fig. 2.4, is divided into 4 bytes with a fixed format and 4 other bytes dependent of the type and code of the ICMP packet. The size of the payload section is variable and also depends on the kind of message.

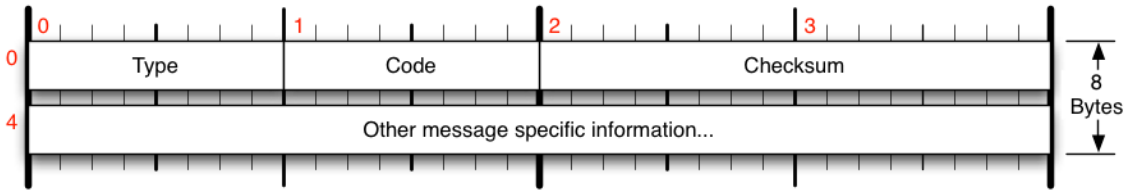



Figure 2.4: Generic ICMP header. Depending on the type and code fields, the message specific information varies. The payload follows the ICMP header.

The ICMP protocol comprehends several types of messages [45] for diagnosing and controlling the IP communication. The type field of the header identifies each one, where the most common are the destination unreachable messages and the echo requests and replies. The code field is a more restrictive classification subsection for within the same type group. For instance, on destination unreachable messages the code identifies the reason for not reaching the target address. Both the type and the code fields characterize ICMP messages using one byte each.

The checksum field enables the integrity verification over the entire ICMP message (the IP's checksum is exclusively applied on its header. The ICMP's 16-bit checksum is calculated according to the Internet checksum specified in RFC 1071 [12].

Conforming to the type and code of the ICMP messages, the remaining 4 bytes of the header take different contents. Not only the header, but the payload section also depends on these fields. In general, the payload of ICMP packets replicates the IP header and some of the data of packets that failed to reach a target host. This fact enables the originator to identify the faulting packets.

### ICMP Features

The main purpose of ICMP is to report errors in the processing of IP datagrams. Such errors occur on a given communication environment between endpoints, precisely on network devices like routers, or in between them, on physical links. They are sent to the source IP address of the originating packet where upper protocols are responsible for inspecting them and act accordingly. The operational flow of ICMP is possible thanks to features particularly designed to control IP communication:

- Reporting protocol. ICMP's objective is to provide feedback about problems arisen during communication;
- Communication control. The information present on the data section is exclusively used for controlling and identifying the relevant problems on the communication path, unlike transport protocols that carry application data;
- Passive identification. There are no guarantees that an IP datagram will be delivered, ICMP simply reports faulting packets without taking any other action;
- Unreliability. ICMP messages may be lost, having no infinite recursive diagnostics of other flawed ICMP messages;
- Integrity control. ICMP packets are verified of their integrity through well-known checksum algorithms.

Naturally, the ICMP protocol retains important mechanisms to error diagnosis, enabling a further wide concept of reliability. It is used to control the communication between network devices where failures of reachability or buffer capacity, shorter routes, and exceeded hops are often advertised. Network administrators take advantage of this to troubleshoot potential network issues by using utilities such as `ping` and `traceroute`.

### 2.1.4 GRE

Cisco Systems developed the Generic Routing Encapsulation, a protocol attempting to be general enough to encapsulate one protocol over another. Essentially, it encapsulates an arbitrary network layer packet into another network layer packet, ready to be routed across networks. GRE operates as a tunnel, meaning that a virtual point-to-point connection is established, much like a VPN. As such, traffic is allowed to travel directly between endpoints, without any interference.

The main purpose of GRE is to enable the communication between peers, that otherwise would not be able to do so over public networks. Therefore, all types of traffic, including *multicast* and *broadcast*, are capable of being encapsulated using a *unicast* protocol, GRE. Routing such traffic between private IP networks across public networks becomes a possibility. As such, the deployment of the GRE protocol brings about some advantages worth mentioning:

- Encasement of multiple protocols over a single supportive protocol;
- Provision of workarounds for networks with limited hops[1];
- Connection of non-adjacent private networks, similarly to VPNs;
- Less resource demand than other tunneling alternatives.

In a more operational perspective, an initial packet, or *payload packet*, suffers an encapsulation by an IP header and a GRE header. The IP header, specifying a protocol type of GRE, acts as the *delivery protocol* to ensure the transmission to the destination. The GRE header identifies the payload packet as encapsulated by GRE, while also specifying its type.

[1] Determined by the number of network devices separating every two networks.

The inner packet stays untouched after the encapsulation process. At the remote endpoint, the packet is de-encapsulated by removing the outer IP and GRE headers. The payload packet remains intact and is now able to proceed to further routing or be processed by any application.

### GRE Header

A GRE packet contains an header section, including the IP and GRE headers, and a payload section with the encapsulated original packet. For this section's purpose, only the GRE header will be described.

The GRE header structure, defined in RFC 2784 [20], along with its possible extensions [18], is depicted in Fig. 2.5. The header consists of 4 mandatory bytes with the possibility for up to 12 additional bytes depending on active GRE flags.

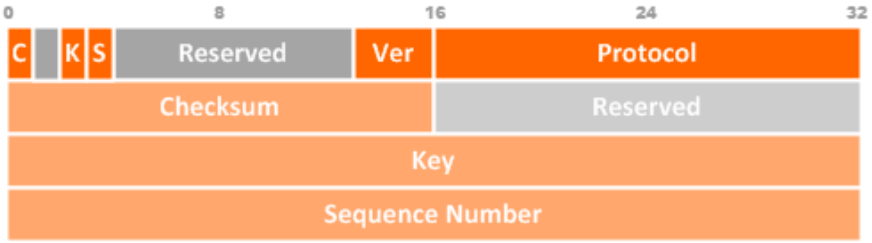



Figure 2.5: GRE header format compliant with RFC 2890. Checksum, key and sequence number fields presence is determined by the GRE flags.

The *C*, *K* and *S* letters are bit-flags that identify the presence of a checksum, a key, and a sequence number, respectively. The key and the sequence number are extensions to the standard GRE. The version field for regular GRE packets is set to 0, for PPTP it is set to 1. The protocol field identifies the protocol of the encapsulated payload packet. The checksum, if present, is calculated over the GRE header and payload. The key optional field identifies the type of traffic and is mostly used during the de-encapsulation of packets in order to permit their expedition to different routing paths. When existent, the sequence number is used to grant reliability on the connection.

## 2.2 Firewalls

Users scattered across the Internet have no guarantee of whom they are interacting with. To provide security from network-based threats, firewalls are placed between the users, on personal or corporate networks, and the outside world. The term *firewall* comes from the literal definition of being a fireproof wall that impedes the spread of fire. In networking, a firewall protects a community of hosts from external incoming danger. Firewalls are therefore **barriers to protect a network** of from unknown and untrusted hosts, providing a single point of entrance by limiting network access through traffic filtering and imposing security policies.

Firewalls are characterized by having two elemental objectives: protection by confinement of hosts and control over host-to-host interactions. Essentially, a firewall is controlled by

a set of rules applied to incoming traffic. Those rules manifest the security policy of the firewall. Protection by confinement aims to provide safety from exterior networks. Despite the advantage of having access to important network services scattered throughout the Internet, liabilities may develop by exposing vulnerabilities. The control over interactions allows not only the creation of authorization and access policies but also content inspection and modification.

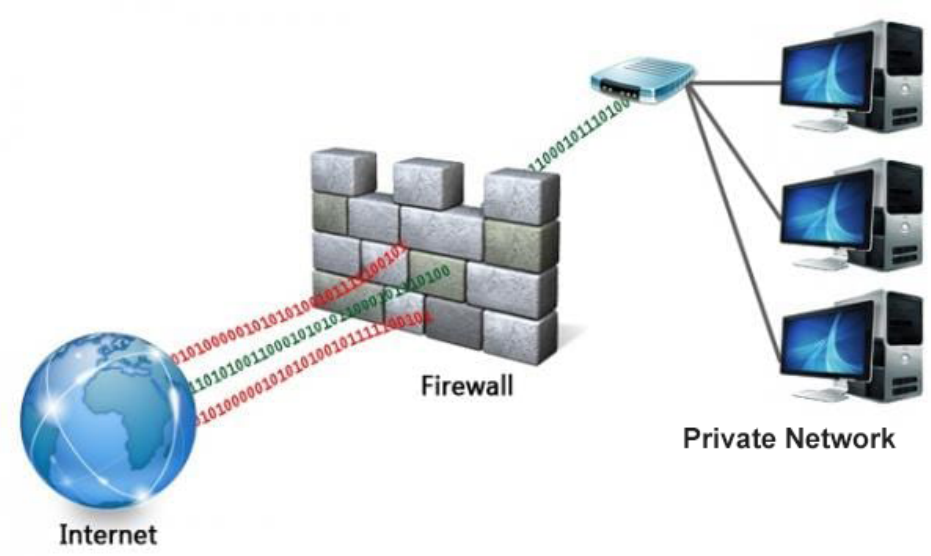



Figure 2.6: Firewall in action. The firewall filters incoming requests based on rules. Authorized requests are able to pass through the firewall into the private network.

A more pragmatic approach identifies three main properties on a firewall's operation. The first is that all traffic, either incoming or outgoing, must pass through the firewall, hence the firewall is a sole point of security and failure. Secondly, according to the local security policy, only authorized traffic may pass through. This grants the ability of access control. Lastly, the firewall is immune to penetration meaning that, if the entrance ports are closed there is no way to break through them, one's best effort is to bypass them.

Initially, firewalls were simply designed to implement a perimeter defense, in other words to guarantee interaction control between the trusted network, as a perimeter to defend, and the other networks. They are called *network* or *organizational firewalls*. It becomes fairly easy to model centralized security policies for such networks. However, despite the previous advantage, it does not suit well for assuring defense in depth. For that goal, other firewalls were designed - *personal firewalls*. This type of firewall does not depend on third-party devices. It applies its own security policies disregarding the network it is connected to. These firewalls are commonly deployed in personal computers to be transparently used by regular cybernauts. Firewalls are also an exceptional location to implement VPNs, since they allow the extension of the security perimeter to other networks and hosts.

Since firewalls stand in front of a network to protect, they may suffer all kinds of attacks, especially if that network is highly valuable. Like all the attacks performed on systems, some are less complex and might produce lesser results while more complex and time-consuming

attacks may even disrupt and compromise connected hosts. Port scanning, IP spoofing[2] and denial of service (DoS)[3] attacks are the most common and well-known, but still remain as real threats for current networked systems.

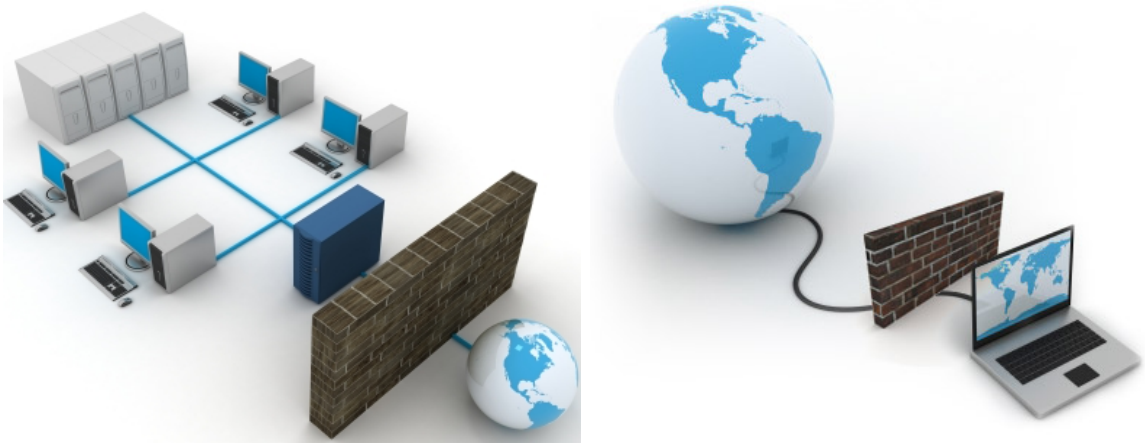


Figure 2.7: Network firewall (left) and personal firewall (right). A network firewall controls access for the whole network. The personal firewall simply protects the user's system.

There are yet more issues that firewalls struggle resolving. IP fragmentation can cause serious DoS attacks due to fragment overlap/overrun or buffer overflow. Malicious code injection is also possible and undetectable by firewalls running IDS systems. Encapsulation is another severe problem where non-allowed traffic is encapsulated into another protocol, such as GRE, which is more tolerable by firewalls. Despite the fact that most of the times the encapsulation is used for tunneling traffic, the previous situation disrupts firewall policies.

One very known application that can be considered as a firewall is the Linux `iptables`. With `iptables`, it is possible to manage personal firewalls or gateways according to sequences of rules contained inside a chain. Whenever a packet reaches `iptables`, it passes through at least one chain for verification and filtering. Similarly to other firewalls, `iptables` accommodates built-in NAT support and permits connection tracking. Adding those capabilities grants `iptables` the classification of stateful packet filter.

According to [13, 67], firewalls can be split into packet filters, circuit gateways or application gateways for their model of intervention, while other authors use different but analogous designations [41].

### Packet Filter

A packet filter is a simpler and earlier version of a firewall. Basically it acts at the network layer, particularly at the IP datagram exchange level. Its sole purpose is to examine packet header fields and decide upon the acceptance or rejection of a single packet. The configuration of the firewall is based on sets of rules that are applied sequentially to each packet. On packet filters, the rules specify actual values of the datagram header fields. Whenever a packet is routed through the firewall, each rule is verified in order. When one of the rules matches the packet, it decides whether the packet is to be accepted and allowed through, rejected with an

[2] Packets with false source IP address for the purpose of identity hiding.
[3] Attack seeking to disrupt resource availability.

error message to the originator or simply dropped. Most of the times, the rule sets end with a default policy of rejecting all packets, which is *those not expressly permitted are prohibited*.

This type of filtering is often called as static filtering since it is based only on information contained in the packet itself, it does not preserve state or notion of connection. It commonly relies on source and destination addresses, protocols, and port numbers. Other filtering based on the direction of the information flow, IP datagram options and ICMP packets is also possible and contributes to increase security against attempts to discover network topologies or obtain information which is otherwise inaccessible.

Despite some limitations, packet filters are simple and easy to configure, fast and memory efficient, and represent the first step to provide an initial line of security to networks and hosts.

#### Circuit Gateway

In contrast to packet filters, circuit gateways evolve static filtering into dynamic packet inspection. While operating at the transport layer, they obtain information not only from one packet but from several packets that depict a connection between hosts. A state is maintained through the usage of variables that determines whether a given packet is starting a new connection, is part of an existing one or does not compose any connection whatsoever. Circuit gateways are therefore designated as stateful or *connection-aware* firewalls.

After authorizing and conditioning connection establishments, these firewalls act as gateways that relay traffic between the endpoints of the connection. Traffic relay can either be transparent to users or not. SOCKS [35] is an example of a non-transparent protocol used for relaying traffic.

Although circuit gateways solve many issues of traditional packet filters only by adding the concept of connection, they are more complex, need additional maintenance and understanding, and may even arise new problems such as DoS attacks on connection state tables.

#### Application Gateway

Application gateway firewalls operate at the application layer and may intercept and mediate interactions from network services. Their key advantage is that they can actually understand applications and protocols, designating them as *content-aware* firewalls. Content-awareness is formally translated into deep packet inspection and modification. This means that application gateways can control all traffic, not only interaction-wise like previous firewalls, but deep into the content of each exchanged packet, bearing in mind the applicational context. Other types of firewalls need to apply security mechanisms to whole packets or whole connections without the possibility to differentiate deeply at the content level.

Concerning the application gateways' implementation, they are realized into a set of applications that mediate each class of traffic. The latter are called as *proxies*. Proxies run specific-purpose code instead of general-purpose mechanisms that have rule sets for all packets passing through. Each proxy serves as gateway that relays traffic to hosts for its specific network service. These mediators allow for user access control, packet inspection and modification, detailed logging and representation of hosts, all due to the specialization for a certain application or protocol.

The need for specialized proxies for every running standard protocol or application stands a

major disadvantage. Not only the previous, but the additional logic for deep packet inspection at proxies also increases latency. Despite reasonable weaknesses, applicational filtering based on processes is much stronger than simple per port or per address filtering. Applicational filtering allows content analysis as well as traffic analysis and solves protocol abuse over encapsulation.

#### Next-Generation Firewalls

Traditional firewalls typically include stateless or stateful packet filtering, network address translation and VPN support. Application-level firewalls add content inspection and modification to traditional firewalls. Next-generation firewalls [38] take the next step by assembling every layer into a single and stronger security element. Their foundational objectives cover, the gain of control over the applications on the trusted network, the scan of those applications, the understanding on which users are initiating the applications and why, and the implementation of convenient control policies to prevent risks or threats.

Next-generation firewalls operate in order to; firstly, identify applications regardless of the address, port or protocols used; secondly, enable accurate identification of users; thirdly, provide deep visibility and control based on policies over applications; and lastly, provide combined protection and resolve on traditional security features.

In short, next-generation firewalls integrate the previous types of firewalls into one and add innovative identification techniques over traffic and users, high-performance, centralized security assurance based on real-time policies, and incorporated intrusion prevention systems.

## 2.3 Port and Service Scanning

The conventional concept of port scanning is rather simple. It is the act of probing a server or host, with normal or forged packets, for open ports and analyze the results. In computer science and networking, a port is an endpoint communication bound to an application, enabling data exchange over the Internet. A port is always associated with an IP address of a host and the transport protocol, either UDP or TCP. Many communications between endpoints may be established since ports are identified by numbers and may vary between 0 and 65535. The associations between port numbers and services are currently maintained by IANA [51, 14].

Since a port typifies an entrance and exit into or out of a computer, port scanning identifies open gateways for every kind of information, that can be either friendly or malicious. Therefore, port scanning can serve as an ethical or non-ethical tool. Ethical hacking generally consists on performing tests to detect security flaws or vulnerabilities on systems, presented with given permission. In this case the vulnerabilities are detected for network administrative purposes. With non-ethical hacking the means are basically the same but the results extend for criminal purposes or personal gains. An analogous representation of detrimental port scanning might be thinking of a thief looking for entrances such as open doors or windows to perform a burglary. Fortunately, the majority of port scans are not attacks and are intended to **determine remote available services**.

Port scanning is a relatively wide concept, considering it integrates two other variations that follow similar methods but obtain slightly different results. One of the variants is *network scanning*, or *port sweep*. As the name implies, several hosts on the same network are

scanned, usually looking for a single specific service running in a centralized way over the network. Another variant is the *service scan*, where open ports are scanned for running services. Contrary to network scanning, service scanning searches for all services running in a set of ports. Most of the times, service scans are executed against a list of non-standard ports in order to identify the services running on ephemeral ports[4]. One simple example is sending a `GET` probe packet to recognize web servers. One last method, not entirely considered port scanning, is the *ping sweep* that simply detects active hosts over the network typically making use of ICMP packets.

The success of a port scanning operation can be distinguished by inspecting the resulting packets. The outcome of a port scanning operation against a given port can be generalized into one of three kinds. Foremost, the reply packet indicates that a service is running, thus translating into an open port. Secondly, the reply packet indicates that no connection is active or allowed, translating into a closed or denied port. Finally, the reply packet is nonexistent implying it was blocked or the sent packet was dropped.

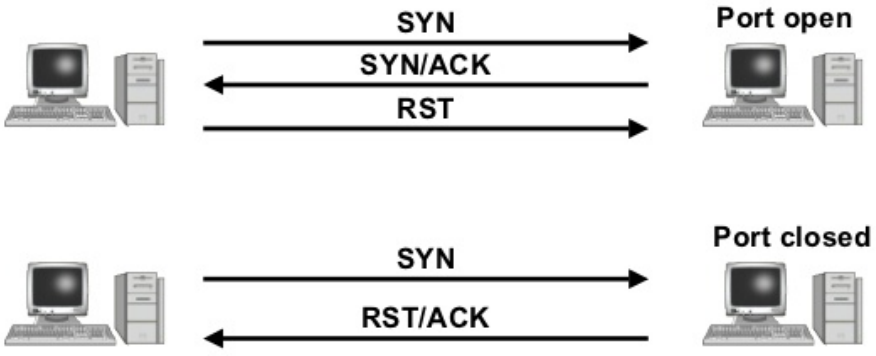



Figure 2.8: TCP SYN scan. The scanner sends a TCP packet with the SYN flag active to be interpreted as an initiation of a TCP connection. The target's response is evaluated for its distinction, resulting in the knowledge of whether the scanned port is opened or closed.

Mainly there are two techniques for performing port scanning onto a server, by using *normal probes* or *stealth probes*. The main difference is that normal probes disregard any consequences resulting from the scanning, while stealth probes are used to maintain the scanning host hidden. Non-stealth port scanning attempts tend to use TCP's three-way handshake mechanism and may come in several types such as TCP or SYN scanning. Stealth port scanning favors the increased difficulty in logging certain packet interactions between the scanner and the target, or even by applying IP spoofing. FIN, X-mas and null scanning are examples of stealth port scanning. A very extensive review on port scanning techniques can be found on [15]. Applications like Nmap[5], Nessus[6] and Foundstone[7] are tools not only for scanning but also for detection and analysis.

A broader concept, directly connected to port scanning, is *stack fingerprinting*, that allows the identification of operating systems on the target machine based on the examination of the stack of protocols revealed on its response packets [67]. Associated with port scanning,

[4] Short-term use of ports in the range [49152, 65535]
[5] `http://www.nmap.org/`
[6] `http://www.tenable.com/products/nessus-vulnerability-scanner`
[7] `http://www.foundstone.com`

attackers can combine the knowledge of services running on the target machine and its operating system to help narrow down the types of exploits. Port scanning and operating system detection may be a strong indicator that a more serious attack may be occurring in the near future.

## 2.4 Secure Channels

In the early days of the Internet, there were no real security threats. This is due to the fact that, by that time, the Internet was more of a micro-system and the amount of users was so scarce that nobody even considered attacking it. Consequently, the Internet Protocol Suite was conceived disregarding security aspects. The succeeding Internet growth, after its deployment worldwide, brought the alliance of the networking and security concepts.

Secure channels are one of many ways to employ security to information traveling along the network. They are inherently associated to cryptography and attempt to conceal it from eavesdroppers on an end-to-end communication. SSH, SSL and IPsec are three of the most deployed secure channels operating in different layers of the TCP/IP protocol stack. The following sections briefly describe each of these protocols for a wide perspective on **securing information exchange over the network**.

### 2.4.1 SSH

Secure Shell, as commonly abbreviated to SSH, is a network protocol that establishes a cryptographically secured connection between two parties for interacting with remote services over unsecured networks, such as the Internet. Despite being mostly used as a remote administration tool, SSH is more universal, and allows for secure interactions over a channel. Remote login, remote command execution and file transfer, SFTP and SCP, may also be employed over SSH. Mainly, SSH creates a secure connection, sometimes referred to as secure tunnel, between a client and a server to enable a protected forwarding of network services' traffic using cryptographic means.

Conceptually, SSH permits any TCP client-server interaction to be secured over the tunnel. This brings some vulnerabilities, mostly verified on organizations where security policy restrictions are applied. Since any TCP interaction can be forwarded inside an SSH tunnel, packet filtering on firewalls would become ineffective, therefore rendering the organization control policies worthless.

In a more functional overview, SSH uses *public-key cryptography* to authenticate both the server and the client. Although password-based authentication is also possible for client users, key pairs tend to be used more effectively due to their increased security. A crucial problem, however, arises from the use of public/private key pairs that is, the reaction upon getting unknown public keys, which may be accepted as valid or discarded as fake. It is important to verify each public key saved on the server by associating it with the identity of each subject that is connecting to the server.

Three major layers identify themselves with the three steps SSH uses to establish a connection between a client and a server [64] before sending any relevant traffic. The bottom layer corresponds to the SSH Transport Layer Protocol [65]. In this layer, the server authentication happens, typically on TCP port 22. The initial setup is made on this layer where strong encryption and integrity protection are provided. The authentication is not based on the user, it is based on the server host using Diffie-Hellman initial key exchange as well as

message authentication and hashing. While public key algorithms are used for the server authentication, *symmetric encryption* is used on the session key for encrypting bulk data transfers.

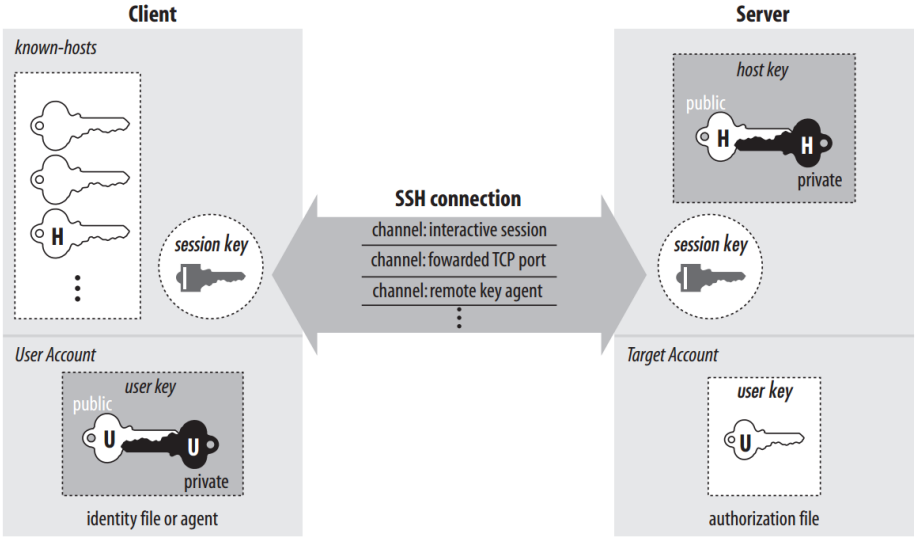



Figure 2.9: Architecture of an SSH system[8]. Both the server and the client possess public/private key pairs to use on authentication. The established connection uses symmetric session keys to encrypt traffic.

On top of the SSH Transport Layer Protocol runs the SSH User Authentication Protocol [62] where the authentication of the user is made using either a password or public/private key pairs. More complex authentication methods may also be used but are not guaranteed to exist in all SSH implementations, while the password and public key are. After the client authentication is performed, this layer provides a single authenticated tunnel to be used for the above layer.

The top layer is the SSH Connection Layer [63]. After the connection is set, it is possible to create several interactive sessions for login and remote command execution. Session channels are multiplexed into a single encrypted SSH tunnel providing full-duplex communication between the client and the server. When all the above operations are complete, the client can initiate encrypted traffic forwarding through a secured tunnel directed to the target server.

### 2.4.2 SSL

Secure Sockets Layer (SSL) is a standard security technology for establishing an encrypted channel between a server and a client. All data transmitted along this channel is assured to be secure and confidential. Therefore, SSL is used to protect sensitive information traveling through unsecured networks meaning that, eavesdropping, data tampering or message forgery is completely suppressed. Most of the times, such information relates to online transactions between web servers and browsers, involving private details of a users' identities. Transport Layer Security (TLS) [17] is a newer adaptation of SSL. It consists of the same security core principles although with significant modifications.

[8] `http://docstore.mik.ua/orelly/networking_2ndEd/ssh/ch03_03.htm`

SSL/TLS provides server authentication as well as client authentication. Despite having the same paradigm as SSH of providing a secure session between intervening entities, SSL has no strict notion of which users are accessing the server. SSL simply identifies both participant classes, the clients and the server.

To be able to create a trustworthy SSL connection, a server is often required of an *SSL certificate*. Only trusted *certification authorities* are allowed to issue new certificates to organizations or individuals, that are audited in advance, and validated of their trustworthiness and legitimacy. SSL certificates contain the requiring entity's information as well as the certification authority's. When in possession of an SSL certificate, a server is guaranteed to be authentic, allowing clients to confide their information when communicating with the server. All the interactions related to SSL certificates are managed by public key infrastructures (PKI) through certificate chains.

The SSL/TLS communication is handled with two sub-protocols. The Handshake protocol performs the authentication between the server and the client and negotiates encryption algorithms and cryptographic keys for securing data, resulting in the formation of a stateful connection. The Record protocol ensures the connection is private and reliable by using a symmetric shared key for the whole session duration.

In short, SSL/TLS is a protocol that provides privacy and data integrity between communicating applications. It is also application protocol independent as upper-layer protocols[9] can layer on top of SSL/TLS transparently.

### 2.4.3 IPsec

The Internet Protocol by itself possesses no inherent security. IPsec [31] is an extension to the IP protocol which supplies data security at the Internet layer. IPsec consists in a suite of protocols that provide data authentication, integrity and confidentiality in an end-to-end security scheme. The security is applied for the whole IP packet, including upper-level protocols. IPsec's main concern is to protect data while in transit, in a lower level, completely independent and regardless of applicational contexts.

Specifically, the suite comprises two sub-protocols, the Authentication Header (AH), and the Encapsulating Security Payload (ESP). The AH protocol implements *authentication* and *integrity* to packets grounded on keyed message authentication codes (see Section 2.5.1). Authentication is ensured by a secret key, while integrity results from data hashing. AH relates both and creates an HMAC instance based on the secret key, the packet payload and the IP header. Due to the usage of the IP header, AH conflicts with NAT mechanisms and requests for NAT-Traversal[10]. The ESP protocol also utilizes the same HMAC instantiation but exclusively using the secret key and the payload. In this case, NAT does not break its specification, but other problems regarding the transport protocols arise and NAT-Traversal still needs to be handled. Additionally to authentication and integrity, ESP also enables for data *confidentiality* between endpoints. Confidentiality is imposed by symmetric encryption, blocking curious middlemen from observing the transmitting data.

Both protocols operate according to specific modes desired for specific purposes. The default tunnel mode encapsulates the original IP datagram as a whole and adds the IPsec's header along with a new IP header. This mode is mostly used for connection tunneling, like

[9] SSL/TLS resides on the application/session layer on TCP/IP and OSI models, respectively.

[10] Mechanism, generally applied to network gateways implementing NAT, that preserves end-to-end IP connectivity.

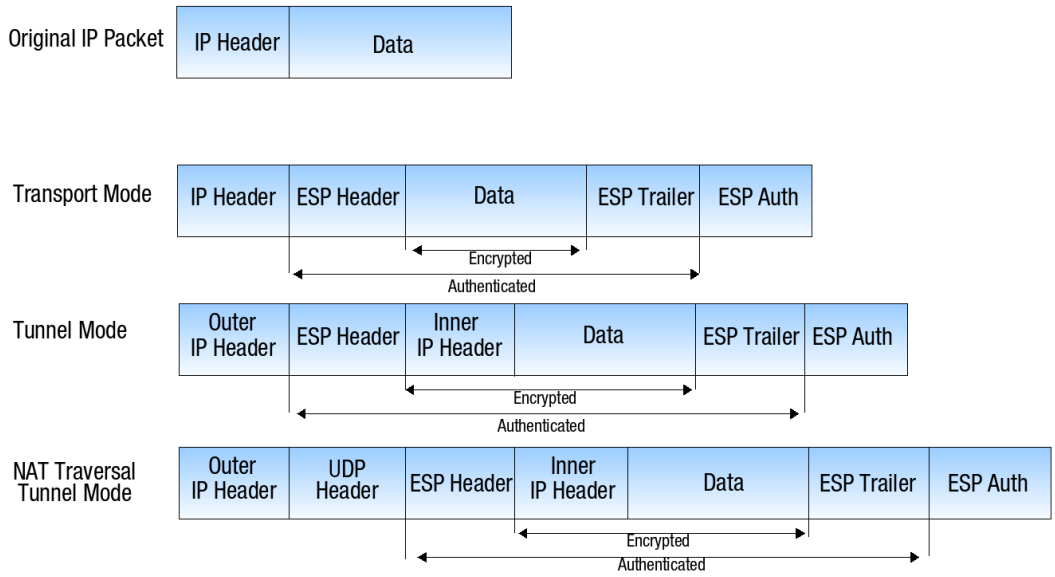



Figure 2.10: NAT-Traversal on IPsec tunnel mode. An UDP header is inserted on IPsec's tunnel mode to assist the forwarding of the packet to the correct services.

---

VPNs, where the need for encryption is applied to a single network hop. On the transport mode, only the payload of the IP datagram is controlled, being most suitable for end-to-end communications between servicing hosts.

The cryptographic keys in IPsec are exchanged either through in-band or out-of-band processes. The keys are exchanged through an Internet Key Exchange protocol [30] or can be manually inserted into both ends of the communication. Security associations also have to be exchanged for IPsec to function correctly. They relate security parameters to traffic flows, making IPsec half-duplex. In short, IPsec creates a boundary between protected and unprotected interfaces for a host or a network granting security over the entirety of IP packets.

## 2.5 Authentication Through Cryptography

Authentication may be accomplished in a variety of ways. In the cryptographic context, symmetric and asymmetric encryption are the most widely deployed methods for protecting important information traveling through the network. Not only they provide data authentication, but also data integrity and even confidentiality. The subsequent sections relate symmetric and asymmetric encryption to forms of message authentication, where **security tokens are used to ensure the legitimacy of message originators**.

### 2.5.1 MAC

Message Authentication Codes (MAC) are small pieces of information that are used as a technique to authenticate messages originated by any sender entity. In other words, MACs are used to confirm that messages are authentic and came from the legitimate sender, and that they have not been changed in transit, either intentionally or unintentionally. They are

cryptographic checksums attached to messages, for proving data authenticity and integrity.

A MAC is a *security token* resulting from a MAC algorithm, that takes a message and a secret key as inputs, hence it is no different than a simple hash except that the compression of the message uses a secret key. The used secret key is actually a symmetric key shared between both the sender and the receiver. As such, a MAC computation uses symmetric encryption for authenticating entities at the receiving side, where messages are verified by applying the same calculations made by the sender.

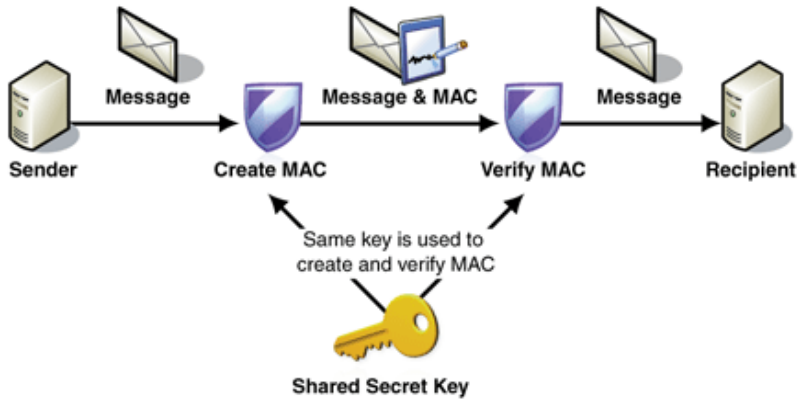



Figure 2.11: Message authentication through the usage of MACs.

Despite similar, security requirements of MAC functions are slightly different to the hash functions'. The security of MACs lies on the fact that it is computationally infeasible to guess other messages' MACs simply by evaluating past MACs, without knowing the key used during compression. Therefore, only legitimate senders may append correct MACs to messages, since they have the knowledge of the secret key.

Although there are several variants of MAC implementations, the keyed-hash message authentication codes (HMAC) [33] are the ones most widely used in security systems. HMAC is simply a recipe for turning hash functions into MACs, thus introducing randomness to MAC computations.

In spite of having considerable advantages, MACs also present some defects. The establishment of the secret key requires for a previous unsecured exchange of confidential information between peers, the key itself. The inability to provide non-repudiation, that is to guarantee a received message was exclusively created by the sender, also stands as a disadvantage. Even though MACs do not warrant for data confidentiality, they do for authenticity and integrity validation, two very important properties for establishing protected network communication systems.

### 2.5.2 Digital Signatures

A digital signature is a mathematical scheme used to validate the authenticity and integrity of messages, just like MACs. Furthermore, it also grants non-repudiation of the originating entity, something that MACs struggle to endorse. Essentially, a valid digital signature assures a recipient that, the message was created by a known and legitimate sender (authentication), the sender cannot deny having sent the message (non-repudiation), and the message was not tampered with in transit (integrity).

Digital signatures can generally be seen as electronic *fingerprints* that associate a signer entity to the message or document it intends to transmit. For that, the signer must be acknowledged with private traits that uniquely identifies him as a valid and authentic individual. In cryptography, the previous mechanism is implemented as public-key authentication.

Public-key, or asymmetric authentication retains the existence of a pair of keys denominated as the private and public keys, where the former is meant to be securely hidden and the latter is made available to the public network. In the context of digital signatures, the private key is securely kept by the signer and is used for signing. Note that, for efficiency purposes, only the message digest is signed. This recognizes that the message was created exclusively by the signer. The message is then appended with the signature and transmitted to the intended recipient. On the receiving end, the signer's public key is used to verify the signature following an identical process, ultimately verifying both signatures of their equality. In case both signatures are equal, the recipient has reason to believe that the digital signature is valid and the message is authentic.

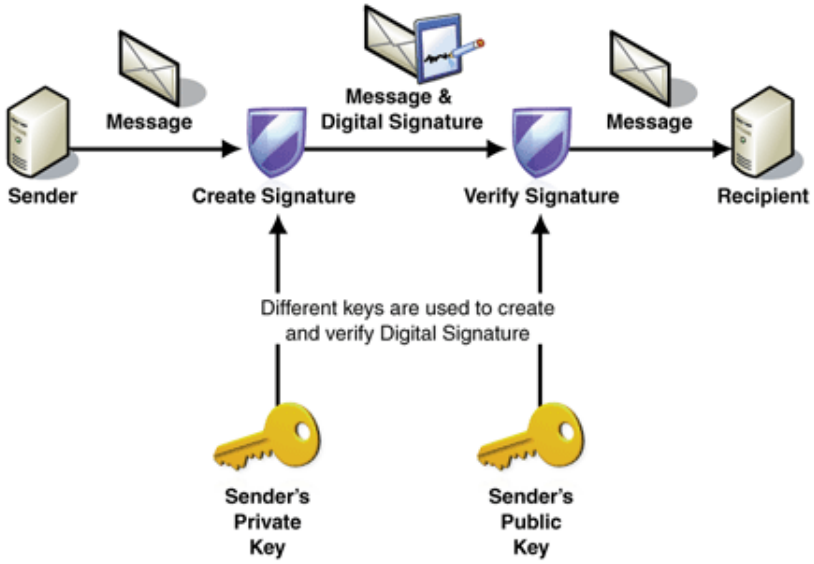



Figure 2.12: Message authentication through the usage of digital signatures.

For further ensuring the trustworthiness of the public keys exchanged through the network, certificate authorities generate digital certificates for key pairs created by entities, according to their verified personal information. Receiving parties are then assured of the credibility of a given public key, simply by applying their confidence in trusted certificate authorities.

## 2.6 NAT

Network Address Translation tries to solve the expected global IP address space exhaustion matter by mapping what usually is an entire private IP address space into another, shorter, public address space. NAT is implemented on devices placed along the path of communication and acts as a default gateway for Internet edge endpoints. In practice, NAT operates on inbound and outbound traffic at the borders of the domain space. Outgoing traffic coming from a local host undergoes a mapping of the local host IP address, to the external IP

address used to preserve the future communication. Incoming traffic uses the mapping table to associate the fake destination address with the real hidden address, in order to forward the traffic to the local host.

The previous one-to-one and one-to-many relations, often related to the concept of static NAT and dynamic NAT respectively, do not stand the most powerful mechanisms of network address translation. The substantial difference through the usage of NAT rests on the possibility to sustain address reuse by **mapping multiple private addresses into a single public address** (see Fig. 2.13). This is characterized as overloading or Network Address and Port Translation (NAPT) as described in RFC 2663 [60]. On outbound traffic, IP masquerading is applied, where the real IP address is masqueraded into a substitute address used only to identify the local network on the Internet. For external hosts, a single entry point is seen. The privacy of the local network is preserved as external entities have no information past the NAT device besides the connecting public IP address. Inbound traffic passes through a port forwarding technique in which the destination port number is associated with an internal host address. For that, the NAT device maintains associations of pairs *<IP address, port number>* instead of simply the IP addresses.

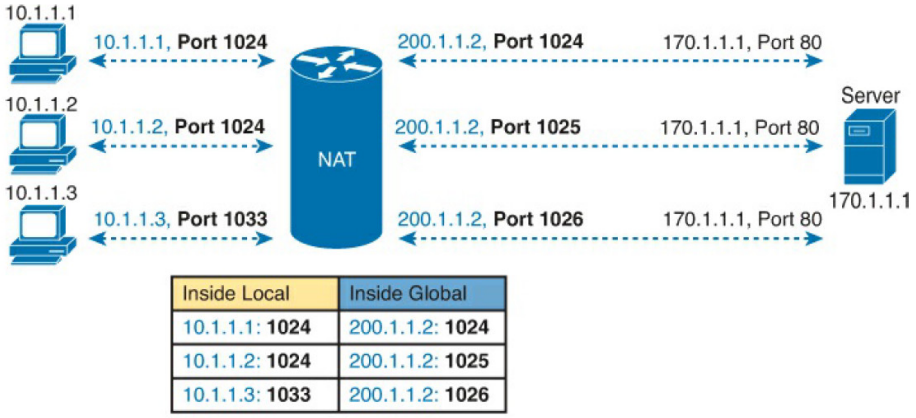



Figure 2.13: NAT Overloading. Private addresses are translated into a single public address indexed by port numbers, in order to establish a connection with the web server.

In order for the NAT mechanism [58] to work, the header fields of the IP datagrams are modified when the translation and forwarding occurs. Since NAT mainly operates at the network level, upper protocols such as UDP [4], TCP [28], ICMP [59] and even GRE must consider potential connectivity obstacles. The end-to-end core principle of the Internet is therefore lost as IP addresses are modified along the path filled with NAT devices. Not only that, but since header fields are modified, integrity checks must be recalculated and reattached to the modified packets, something that happens on UDP and TCP transport protocols. As applications negotiate some of the header fields at the application level, NAT implementations must be aware of upper layer protocols and need to conjugate with Application Level Gateways to perform NAT traversal correctly, examples being FTP and SIP protocols. NAT further stands in the way of IPsec tunneling mechanisms of integrity checking.

The translation methods for NAT implementations may be classified into one of four categories. On full-cone NAT, one internal address is mapped to a single external address, making it possible for external hosts to communicate with the internal address by connecting to the external address. On address-restricted cone NAT the same principle applies except

that the external host must be contacted first for it to be able to connect to the local host. Port-restricted cone NAT narrows the restriction down to the specified port instead of the IP address only. On symmetric NAT each request from the same internal IP address and port number is mapped to a unique external pair address. In real world NAT behavior, these methods are often combined together to more specifically pertain the actual NAT traversal method.

## 2.7 Access Control

Access control is a conceptually wide mechanism that exists to limit the interactivity between any foreign user and the resources of a secured system. In this way, it prevents injurious activity that could lead to security breaches of said system. Some basic principles and practices are described in [56] and an extensive overview on access control is provided in [53]. Access control grants the ability to control, restrict, monitor, and protect resource availability, integrity and confidentiality. The foremost challenges access control faces are the variety in types of users accessing the system and the variety of resources and their authoritative privileges. The *subjects* represent the users that can perform actions on the system and the *objects* represent the resources which's access is controlled from subjects. The *access* is the flow of information between subjects and objects.

For its broader definition, access control is a 4-step mechanism that includes **identification**, **authentication**, **authorization** and **audit**. Most of the times some of these steps become agglutinated and are performed together. On less complex systems sometimes it is possible to merge the authentication and authorization steps while the audit may not even be present. On composite systems the 4-step mechanism is very well defined and delimited where each step is assured by independent processes. In a narrower definition, access control comprises only the access approval based on authorization policies and leaves the authentication completely apart and independent.

The first step for controlling access to a requesting subject is the identification. While most times being combined with authentication, identification itself consists on the process of verifying that the identity the subject provides is bound to the entity it claims to be. It does not however represent the actual authentication. This is accomplished through *identity proofing* where the identifier of the subject must be unique.

In contrast to identification, authentication does not refer to the act of stating the subject's identity. Authentication, which is the second step of access control, is the process of actually confirming that identity. For this step, not only the identity of the subject is needed but its accessing credentials must be supplied to verify its legitimacy. Subjects may be authenticated in several ways that generally fall into one of three categories, based on what are known as the factors of authentication:

- Knowledge factor - represents something that the subject knows: password, PIN, challenge-response, security question;
- Ownership factor - represents something that the subject has: ID card, access card, hardware or software token;
- Inherence factor - represents something that the subject is: biometrics, signature, voice.

Typical security systems offer single-factor authentication. This type of authentication does not offer much protection against intrusions since there is only one mechanism attackers have to bypass in order to gain access. When added another authentication mechanism, attackers must circumvent two authentication methods that are totally independent. This results in an increase of security usually performed on more complex high-level security systems. Another factor considered nowadays is the location factor that represents where the subject is. The abstraction of the location factor is specified in the context of computer security as the network location of the subject, which can be a user inside a VPN or past a firewall.

Authorization is the step of access control that specifies the access rights to resources. The authorization step assumes authentication has been successful and the requesting users are considered legitimate. In order for the authorization process to work, two phases are executed in order; definition of policies for authorized access, and enforcement of policies for approving or disapproving access requests. On the first phase, the owner or administrator defines the policies for access to the resources. The second phase happens when an incoming request from a user is detected. The request is verified against existing policies taking into account the requesting user and the requested resource.

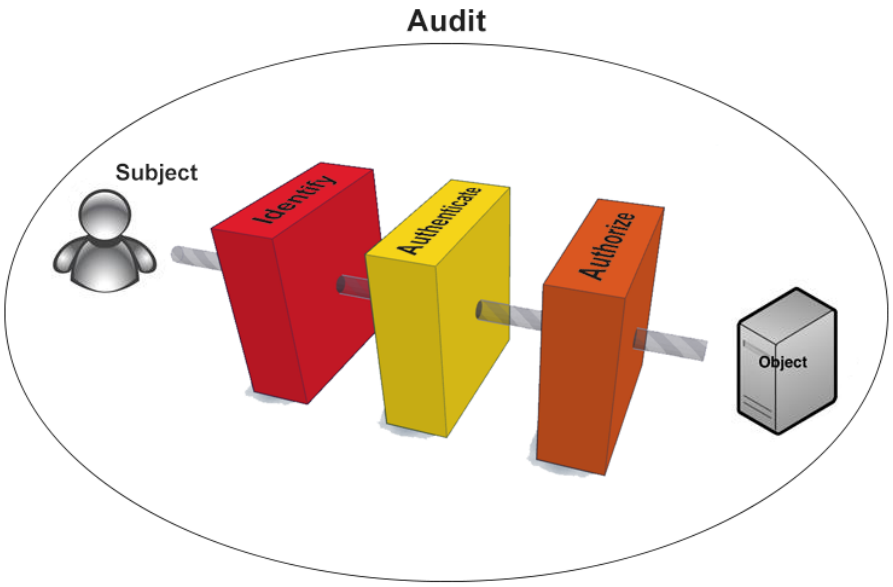



Figure 2.14: Four-step access control. The subject requesting access to a resource must pass through identification, authentication and authorization processes. Auditing is enabled as an administrative tool to detect possible system failures.

Systems may implement access policies in the form of *access control lists* (ACLs) or based on *capabilities*. ACLs are specified for each object. The policy for the object is then represented in a form of list that contains the subjects and their accessing rights. Since ACLs are object-related, reviewing or revoking accesses to an object is fairly simple. Capabilities are subject-related and suggest a complementary approach. Each subject possesses a capability list indicating its accessing rights for every resource. In order to fully understand the previous concepts, an analogous real life access control situation would be for a given individual trying to access a convention. The ACLs would translate into a list, retained by a receptionist at

the entrance, where access is granted by name. On the other hand, the capabilities approach would introduce tickets that the invitees would have to present. ACLs can therefore be associated to what or who one is and, in order to change the accessing rights, the controller must change the access list. Capabilities identify what one possesses, such as a key. This allows the key to be exchanged between subjects.

Modern operating systems implement authorization policies as variants of the basic access types, read, write and execute. These permissions vary according to the model that supports the system. Generally, there are three access control models that represent the majority of running systems. The Discretionary Access Control (DAC) and the Mandatory Access Control (MAC) models were the first to be conceived. The third access control model is the **Role-Based Access Control** which subsists on both DAC and MAC models making it slightly more complex and powerful.

Auditing is the last step of an access control system. While sometimes being neglected, it offers such systems the possibility to record and log every interaction, enabling the detection and eventual prevention of security threats and intrusions, much like IDS and IPS systems do.

### Discretionary Access Control

Discretionary policies dictate the access of users to information based on the user's identity and its authorizations for each object. The access rights and permissions are set up by the owner or creator of the data. He is responsible for determining which users have access to specific resources. Each request arriving at the system is checked against the user's authorizations and access is granted if the requested authorization is allowed by the owner.

Discretionary policies however do not have any sort of control over the intentions of the users requesting the information. Since no mandatory actions are taken, the subject can deliver the contents of the received information to unauthorized parties. Content distribution over unauthorized entities stands a problem for discretionary access control.

### Mandatory Access Control

Mandatory access control sustains a complementary approach of the previous method. In this method, subjects and objects are assigned security levels. The security level of an object reflects its information sensibility while the security level of a subject reveal the trustworthiness of the subject. The dissemination of information is now based on the idea of trusted subjects.

When requesting access to a piece of data, the subject's clearance and the objects security label are verified and if they match the subject is granted access. Not only the subject is allowed to access information at the same security level of its clearance level, but it can also access information hierarchically above or below depending on rule models such as Biba[11] and Bell-LaPadulla[12]. Implementations combining properties of both methods are better suitable for security systems. A detailed overview of mandatory access control is considered by Sandhu in [54].

[11] Any user can read up and write down, data integrity is maintained.
[12] Any user can read down and write up, confidentiality is maintained.

### Role-Based Access Control

Two extremes were created by mandatory policies being too strict and discretionary policies being too tolerant. The need for a more commercial solution sits at the base of the development of role-based access control methods that couple properties of both previous methods.

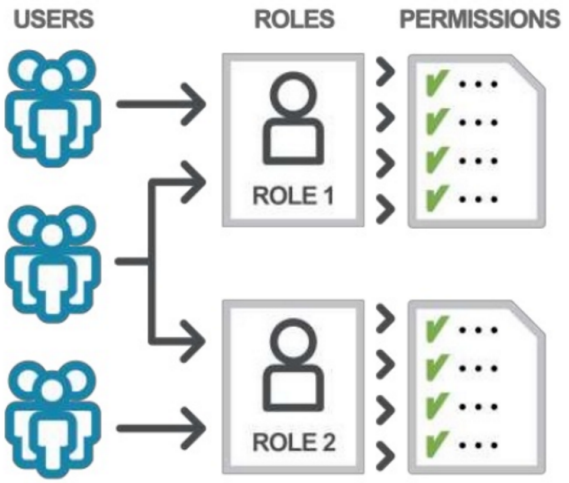



Figure 2.15: Role-based access control. Users are assigned roles consisting of specific permissions on the system.

Role-based policies rise [55] to regulate users' access to information based on the activities they perform on the system. The permissions to certain activities are associated to roles, and users are assigned to the appropriate roles. This facilitates the access rights appointment. Instead of specifying the permissions for each subject-object pair, the user is simply assigned to a role according to its job function, responsibilities and qualifications. Users may be reassigned from one role to another. Roles can also be granted new permissions, and permissions can be revoked as needed. Roles stand as a group of subjects that inherit the same privileges. This naturally reduces the need for an extensive list of user permissions.

Some properties are inherited from previous access control models. Roles can be placed in a hierarchical structure where relations between their upper or lower siblings are present. Roles also implement the least privilege principle where subjects are only capable of doing exactly what they were intended to do according to their activities. Lastly, users cannot have enough privileges to misuse the system on their own, separation of duties must exist.

## 2.8 Virtual Filesystems

The origin of virtual filesystems is essentially based on the demand for uncomplicated user access to kernel information. A virtual filesystem is simply an interface that abstracts the communication between the kernel and user applications. It can be viewed as a window or a snapshot of the kernel's information during runtime.

In order to exchange data between user space and kernel space, the Linux kernel provides some useful RAM-based interfaces. The Linux development follows a paradigm of *"Every-*

*thing is a File"*, as such, the communicational interface is materialized into several virtual filesystems, where all kernel parameters and configuration values are available through virtual files. By means of standard read and write functions, user space programs can access these files in a transparent and bidirectional manner.

Despite offering rather similar functionalities, the different virtual filesystems are designed for distinct purposes. The following sections identify three different approaches for **user applications to interact with the Linux kernel**, the well-known *Procfs*, the new sucessor *Sysfs* and its complementary version *Configfs*.

### 2.8.1 Procfs

*Procfs* [32] is one of the most dated virtual filesystems in Unix. It is a special filesystem that gives access to information regarding each running process. The information is granted as snapshots of the processes' status, structured as an hierarchical file-like view according to process ids. The *Procfs* filesystem is mounted on `/proc`, visually resembling a regular directory. It acts as an interface to internal data structures in the kernel, to obtain runtime system configurations. For this reason it can be regarded as a control information center for the kernel.

The *Procfs* filesystem provides a method for user programs to access kernel space information. Many developers take advantage of this fact and use this virtual filesystem on applications connecting to the kernel. In fact, many utility tools, such as `lsmod` and `lspci`, are simply applications that read files present on `/proc`.

Due to its high availability and handiness to both the user applications and the kernel, *Procfs* has been overrun with nonessential information becoming a dumping ground for a whole range of system data. Not only that, but it also gained write permissions[13] allowing for unstructured definitions on the `/proc` directory. *Procfs* stands as a center for legacy system information thanks to new implemented and active solutions.

### 2.8.2 Sysfs

*Procfs*'s deficiencies led to the development and introduction of a new virtual filesystem on the Linux kernel, the *Sysfs*. *Sysfs* [39] is a pseudo filesystem very similar to its predecessor when regarding its functionality. It exports information about kernel subsystems, hardware devices and other device drivers to the user space through virtual files. The *Sysfs* filesystem features a well-defined structure and strict directory hierarchy based on the internal organization of kernel data structures. The file definitions follow precise rules. Only one value per file is allowed for representing a single configuration parameter, thus granting a higher level of abstraction to installed drivers.

*Sysfs* is a bidirectional interface for representing kernel objects, their attributes, and their relationships with each other. It can express itself as a kernel interface for exporting the previous items via the virtual filesystem, or as a user interface to manipulate the same items that map back to their representations as kernel objects. As such, handling configurations for kernel structures becomes an easy task by simply reading and writing to a file present on the `/sys` directory. *Sysfs* relies on a uniform structure with inherent guidelines for developers to follow, making it a valuable interface when communicating between the kernel and user applications.

[13] `sysctl` allows manipulation of kernel configurations at runtime. In *Procfs* it is implemented on `/proc/sys`.

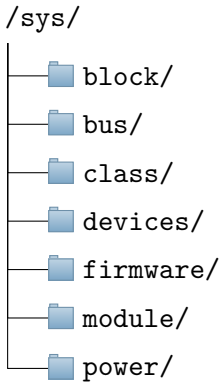



Figure 2.16: Top-level *Sysfs* directories. The directories represent the major subsystems created at system startup. After their initialization, each one begins to discover objects respecting its system scope.

---

### 2.8.3 Configfs

*Configfs* [8] is yet another RAM-based filesystem implemented by the Linux kernel. Despite being similar to *Sysfs* on a functional level, they are in fact different and complementary. Where *Sysfs* is a filesystem based on the viewing and manipulation of kernel objects created on the kernel side, *Configfs* is quite the opposite, a filesystem manager for doing the same operations, but on **kernel objects created by user applications**.

*Configfs*'s operational description might induce to a violation of the core principles on kernel-user interaction, yet this is not the case. Kernel objects are obviously created by the kernel itself, however, the user applications are in full control of the life cycle of such objects. This means that user-space programs create, manipulate and destroy kernel objects at will, something that does not happen with *Sysfs* where the user applications can only perform value manipulations. Object definition and syntax follow the same principles as the *Sysfs* filesystem.

User applications are allowed to create configuration items[14] simply by issuing a call to a `mkdir` operation. The destruction of these items is conversely possible by using `rmdir`. Once an item is created, its attributes become ready to be modified by simple `read` and `write` operations in their many well-known forms. Attributes may as well be grouped together just like in *Sysfs*. The items and groups are then registered into a subsystem, representing a client module and appearing as a subdirectory under the top-level `/config` directory.

[14] Equivalent for kernel objects in *Sysfs*.

# Chapter 3

# Related Work

On the following sections, the related work is extended into different solutions that solve each singular system's basic objective. It is divided into solutions for port hiding and system concealment, and techniques for controlling user access on authentication and authorization.

## 3.1 Controlling Port Scanning

Concealing the system is most often a synonym to averting port scanners. This section describes different solutions used to subvert port scanning. Although their main purpose is port hiding, these solutions also expand on user authentication, which will be addressed further below on this chapter.

### 3.1.1 A TCP-Layer Name Service

In the current days, hosts can be scanned for open ports to detect running services through the usage of well-known applications called *port scanners*. This is made possible due to the fact that ports are identified by a relatively small number. In his paper, Freire [23] proposes a name service for TCP ports that enables them to be reached by a name instead of a number, while still providing regular means for non-complying or legacy hosts. The transport layer name service evolves the standard *three-way handshake* by integrating name resolution and user authentication on TCP's reliable and bidirectional communication. The solution rests on the purpose to grant access only to well-intended users, thus it preserves the concern that attackers can watch and intercept traffic at will. Two name resolutions models, that annul port scanning techniques by identifying ports via naming, were suggested.

The Simple Name Resolution Model handles no user authentication or authorization. It merely provides name resolution over the TCP's connection establishment process. When the server receives a `SYN` segment with a port name, the name service resolves it into a port number and the association is sent back to the client in the `SYN+ACK` segment. Thereafter only port numbers are used to maintain the connection. The port name is sent on the payload with a TCP option identifying the use of port names.

The Enhanced Name Resolution Model expands name resolution onto external logic processors called *Domain of Interpretation Resolvers* which keep closely integrated with the TCP protocol. Association registry, validation on the client side and revalidation on server side are the primary functions of DOI Resolvers, called asynchronously during the three-way hand-

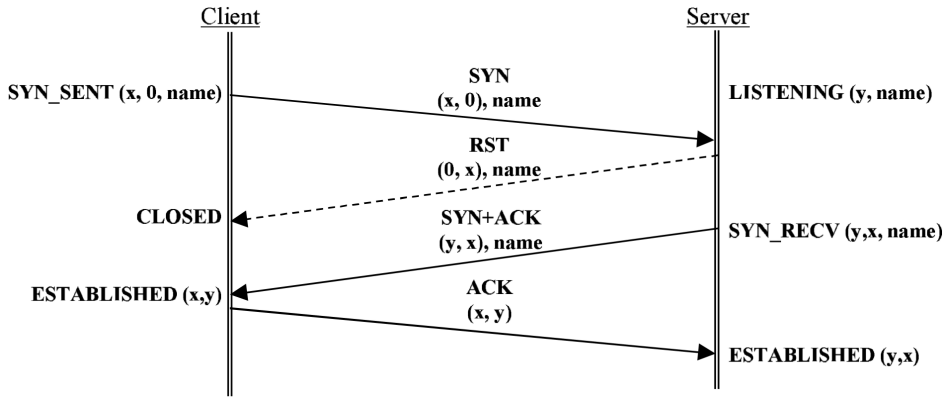



Figure 3.1: Simple TCP name resolution. The standard *three-way handshake* is extended to support port names.

shake. This extended model grants secure access via pre-shared keys between endpoints. Diffie-Hellman key distribution is also part of the data exchanged during the connection establishment.

### 3.1.2 Port Knocking Mechanism

Port knocking is a mechanism that uses connection attempts on a set of predetermined closed ports to implement network level access to services. Only credible users that generate a correct authentication sequence may automatically and transparently provoke a dynamic action over the accessed host or network. Generally, port knocking takes place on client-server models where a port knocking daemon stands at the server end intercepting and logging received packets. After a correct sequence of ports originated by a client, the daemon allows it to connect to the wanted service port or execute a remote command. Martin Krzywinski was considered to have formally pioneered the port knocking mechanism [34].

Connection attempts may be in the form of UDP or TCP `SYN` packets carrying the authentication information. A later version [49] using ICMP packets for port knocking sequences was identified as conceivable and implemented shortly after. The authentication information is comprehended on the packets headers where the IP address, the destination port number, the time and checksum fields are used to encode the knock sequence in order to dynamically modify firewall rules according to the client's IP address, requested service and duration. A deviation of port knocking, single packet authorization [49] combines all the authentication information on the payload of a single packet, maximizing throughput and efficiency, while also adding supplementary security features.

The vital purpose of port knocking is to prevent an attacker from scanning the secure host for open and potentially exploitable ports. Port knocking is a stealthy authentication mechanism since all the ports are closed until a correct sequence of connection attempts appears while discarding all the incorrect ones. Due to the fact that port knocking introduces random port numbers as correct sequences, sequential and targeted port scanning tools become ineffective. Nevertheless, traditional port knocking is presented with several security flaws that can be abused by ambitious attackers [37]. NAT-knocking describes the situation where two clients behind the same NAT share the public address, therefore it becomes impossible for

a port knocking system to differentiate them, allowing for one of the clients to be a potential attacker. DoS and DDoS attacks against memory buffers that support each connection attempt are also possible. Replay attacks also represent a major issue for conventional port knocking.

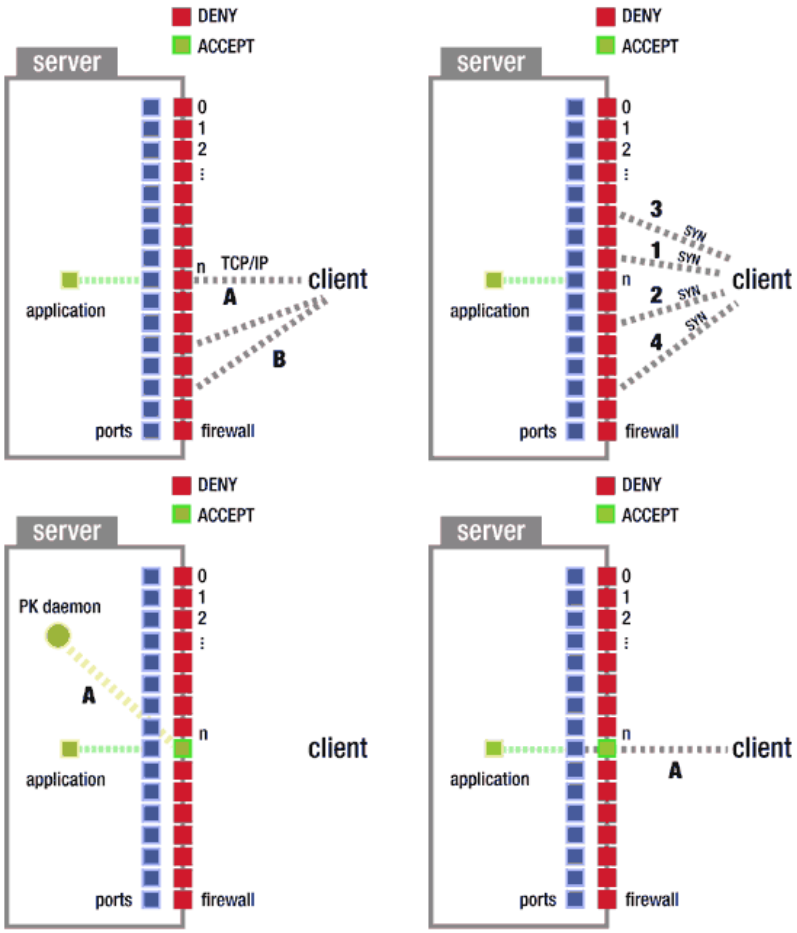



Figure 3.2: Port knocking mechanism[1]. Four-step process in which the client is permitted to access a port running the desired application, only if it provides the correct knock sequence.

Several subsequent implementations were created to solve most of port knocking's weaknesses. In [2] the authors overview such implementations and present their advantages and disadvantages while also giving a new solution. It is based on source port sequences. Instead of producing a sequence of packets destined to specific ports on the remote host, the originating user sends a sequence of packets to a single remote port where the source ports are used as the information for authentication. Each user is individualized for his distinct source port sequence and he is verified against a white-list of users. This article alleges to solve port scanning since it is impossible to associate traffic of newly opened connections to a previous sequence of packets. Therefore, an attacker cannot infer on the life cycle of a given service preventing replay attacks. Source NAT deployment, however, disrupts this system's implementation.

[1] `http://www.portknocking.org/`

An early implementation described in [5] uses a challenge-response method to authenticate users alternatively to one-time passwords due to time constraints and simplicity. It characterizes itself as a lightweight NAT-aware solution that focuses on solving disordered packets and replay attacks. Note that the replay in this case is solved using encryption which is much more effective. The authentication can be mutual or unilateral only. An innate side effect of the challenge-response method is that the port knocking system exposes its existence to attackers.

A relatively recent implementation [1] proposed a more complex and heavyweight approach that includes steganography[2] and mutual authentication to the port knocking system. The authentication information is handed over on the payload of the packets in the form of an image. After the image processing, a mutual authentication occurs using key encryption that will eventually open the required port or execute a remote command. DoS-knocking and replay attacks are inhibited thanks to built-in intrusion detection and encryption, respectively.

### 3.1.3 Lightweight Concealment and Authentication

Attacks on a network connected system depend mostly on the information collected from such system. The information is naturally collected through specifically designed probes injected on the network or simply through the observation of network traffic between hosts. Existing techniques to protect network connected machines tend to rely on packet filter firewalls, that filter unwanted traffic at a very low level disregarding application-level exploits, or application-level security, that usually perform heavy cryptographic calculations enabling other types of attacks such as DoS attacks. Another possible solution to this problem stands on the lightweight concealment of the ports used to provide network services to authenticated users from the unauthorized attackers.

In a work [6] published by Intel, the authors argue that there is a need for computationally cheap and simple security mechanisms that allow the early abandonment of the majority of unauthorized attempts. It also claims that services should be hidden while providing no response whatsoever to subvert unauthorized port scanning. Generally, the paper proposes a lightweight authenticated one-way signaling mechanism that augments the function of stronger authentication mechanisms like IPsec, SSL, SSH, and more. Comprehensively, it identifies three main properties:

- Services should be hidden, meaning that the end hosts should be considered as stealth devices that are invisible to other devices except for authorized entities;
- Access credentials should be easy to validate, yet difficult to falsify, meaning that invalid credentials should be discarded in order for the service to remain concealed;
- Application security mechanisms should not cease to exist as they are still used to provide true end-to-end authentication between services.

In a more practical manner, the paper describes three variations (see Fig. 3.3) of the same basic idea which are implemented at the endpoints. The implementations are intended for TCP but the authors allege that the same mechanisms can be analogously extended to UDP.

SSTCP, or Spread Spectrum TCP, makes use of several TCP `SYN` segments to send an authentication key from the client to the server. The key is modulated and encoded on a

[2] Technique used for concealing a file within a file.

TCP header field in each `SYN` segment transmitted. There are two viable options that may resist end-to-end communication disrupters, which are the destination port number and the initial sequence number. For SSTCP, the destination port number was chosen since the initial sequence number may be altered by proxies. When a client tries to start a TCP connection, before the three-way handshake, the `SYN` segments encoding the key are sent to the server. This resembles the port knocking mechanism. Once the server decodes and obtains the key from the received packets, the key is verified and the client is authenticated. The normal three-way handshake process may begin on the next packet sent by the client. A noticeable handicap is that the key must be previously shared between both parties.

In TGTCP, or Tailgate TCP, authentication is made on a single packet transmitted from the client to the server containing a hashed key alongside other authentication parameters used to inhibit replay attacks. Such parameters are: the current time, the client identifier, the source public IP address, the destination address and port number, and a hash of all the previous parameters concatenated with the shared key. This packet must be followed closely by the initial three-way handshake packet due to the fact that, once the key is verified as legitimate, the server's firewall opens the corresponding port for a very short period of time, typically two seconds. Only within that timespan the attempts for the client to initiate a TCP connection are allowed. This solution stands imperfect for its time related constraints, bearing in mind the tendency of network delays over the Internet.

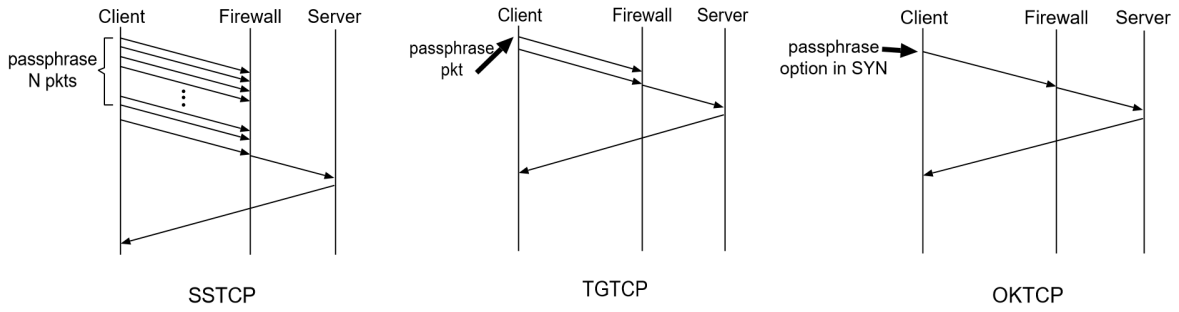



Figure 3.3: SSTCP, TGTCP and OKTCP authentication mechanisms.

The last implementation proposed is OKTCP, or Option-Keyed TCP, where the authentication parameters are included on the actual three-way handshake packet. The authentication parameters correspond to the same used in the Tailgate TCP and are encoded as an IP or TCP option field. The drawback of this approach resides on the relatively small limited size of the option to encode the shared key.

These approaches present various benefits on lightweight authentication and network service security. There is a reduced opportunity for attacks since exploits are less likely to occur on endpoints that do not appear to exist. There is also a reduction in vulnerability to attacks since the attacker must bypass these new mechanisms. The rejection of attackers is turned into a lesser computational cost due to the lightweight verifications, which stands a major benefit.

## 3.2 Exercising Access Control

As stated, access control typically comprises four fundamental steps: identification, authentication, authorization, and audit. Identification and authentication are frequently employed together on security systems. In light of this fact, the solutions described are combined simply into an authentication section. Authorization is generally implemented as a standalone mechanism, independent of authentication. The authorization section focuses on solutions that use the different methods for authorizing users, completely assuming the authentication process to be handled previously by separate mechanisms. For the purpose of this dissertation, audit solutions are not discussed.

It should be noted that there are several other solutions for both authentication and authorization but only these were mentioned, since they are potentially interesting to the proposed system.

### 3.2.1 Authentication

#### 3.2.1.1 User-Based Access Control Framework

Packet filter firewalls essentially inspect the header of IP packets and decide upon their acceptance or rejection through the firewall. It is a simple mechanism that provides a first level of access control based on host information. IP packets however cannot be considered trustworthy solely based on the host's information, namely the IP address. Packet filters do not provide any type of security based on actual users' information meaning that, impersonating hosts becomes relatively easy for moderately motivated attackers performing IP spoofing techniques.

In this solution [68], a framework for enforcing user-based access control over packet filter firewalls is described. The framework intends to prevent unauthenticated traffic to reach a protected host by adding extra information to packets, the originating user's identity. This enables the packet filter to survey filtering rules based on user identity. The strategy used sits on the single sign-on[3] concept layered at the network level.

| | | |
|---|---|---|
| 1 | C → S | : packet |
| 2 | FW→ C | : no source UId, S, IdP |
| 3 | C →IdP | : user authentication |
| 4 | IdP→ C | : AIP, PAK, AIT |
| 5 | C →FW | : UId registration, AIT, AA |
| 6 | FW→ C | : UId registered, AA |
| 7 | C → S | : packet, AA |
| | FW | : AA's AIP → PAK, UId |
| | FW | : Validate AA with PAK |
| | FW | : check UId access |
| 8 | FW→ S | : packet |

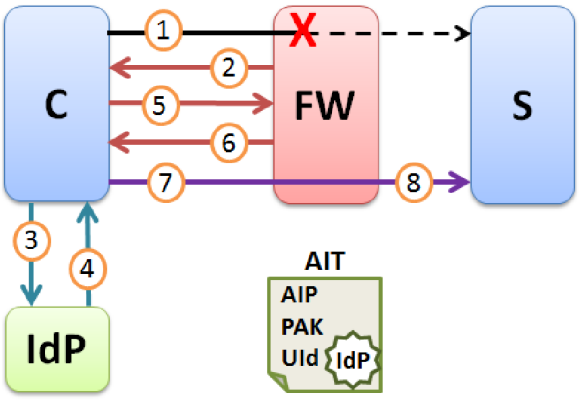



Figure 3.4: High-level protocol for enforcing identity-based access control.

User authentication is set up based on a set of *credentials* that identify the originator of a

[3] Access control mechanism where a user generally accesses several systems by only logging in once.

packet. Whenever a packet is received at the server-side firewall it is checked for an identity mark. If not present, the server sends back an error indicating that the user must first authenticate himself to an Identity Provider trusted by the server. After the user has proven his identity to the identity provider, he then uses the received authentication token to register himself on the server. At this moment, whenever an identity marked packet is received at the firewall, attached with an access authenticator that includes the user's identity, the firewall checks if the packet truly originates from the user by verifying cached registrations and allows it to pass through to the server. The access authenticator marking uses a new 24-byte IP header option that may potentially create coexistence problems. This whole process grants the ability to enforce access control policies on the firewall, not only per packet but with a user identity associated with it.

#### 3.2.1.2 Challenge-Response Authentication Mechanism

The challenge-response authentication is a mechanism familiar to everyone as it is a broad and universal concept employed worldwide, not only in computer security but also in every other subject where authentication is inherently present. By definition, challenge-response mechanisms are applied where verification of a given entity is needed when trying to access something that is controlled or supervised by another entity. The controlling party presents a question - *challenge* - at which the accessing party needs to provide a valid answer - *response* - to be authenticated. The simplest application of a challenge-response mechanism is ordinarily known as password authentication. The challenger asks for a secret that the responder must supply correctly in order to be successfully authenticated and granted access to the protected resource.

Regarding computer security in depth, this authentication method becomes an increasingly more complex matter thanks to the advance in network communications and Internet in general. Automated scripting machines for eavesdropping and attacking traffic are almost always accompanying every network connection. Despite the simple method above being implemented in some systems, it does not grant any security value, making the systems attend severe security issues. Therefore, challenge-response mechanisms have very different methods of implementation, where the most commonly used in communicational security are the cryptographic techniques.

Fernandez and Warrier [22] describe a composite pattern to achieve secure remote authentication and authorization for distributed systems based on two patterns, proxy server and role-based access control. When a requiring user wants to access protected information a certain flow of steps representing the pattern is observed:

- The authenticating user makes a request for a network service through a proxy server representing a single entry point;
- The request is routed and forwarded to the remote server containing the needed authentication information;
- The remote server responds to the client by sending an access-challenge back, passing through the proxy;
- The client calculates a response for the challenge and sends it to the proxy that again mediates the traffic and forwards it;

- If the response matches the expected answer for the challenge, the remote server replies back granting access to the client, which is, as of this moment, authenticated.

After the authentication process, the user is still not allowed to access the protected resource. An authorization operation must follow through. The client requesting for a specific resource is authorized based on the role he possesses at the remote server. If such a role grants permission of the given client accessing the requested resource then, the remote server acknowledges this information back to the client along with the effective access to the protected resource.

While the challenge-response authentication mechanism is enforced on many known protocols: RADIUS, SCRAM, LDAP, SSH and more; it is also used for its security capabilities on systems and in general applications. In this article [61], the authors implemented an authentication solution based on the optical perception of cameras. It uses OpenID single sign-on for enabling authentication over several network services using the same user authentication information. The user trying to access a web page is presented with a challenge in the form of a QR code. The user captures the QR code and a response is calculated in the same form. The server verifies the correctness of the response using a secret key, previously shared with the user. The challenge-response paradigm is invoked in this solution as a fast, simple and secure authentication mechanism that uses encryption.

#### 3.2.1.3 Public Key Authentication

Asymmetric cryptography grants integrity, authenticity and confidentiality. Public key, or asymmetric, cryptography gets its name from the conceptual aspects behind it. It uses a pair of different keys identified as the *public* and the *private* key. Evidently, the public key is intended to be distributed along potential communicators while the private key must be kept hidden and protected only accessible by the owner. Mainly there are two possible ways of exchanging messages between entities. Either the receiver's public key is used for encryption ensuring only he can decrypt the message using his own private key, or the message is encrypted using the issuer's private key convincing the receiver of the message's originator. The strength of a public-key cryptosystem lies solely on the security and protection of the private key, given that the generation of the key pair withstands a high degree of difficulty on the discovery of the private key using the public key.

One of the main problems relating access control and public key authentication is the association between public keys and identities. A remote user authentication scheme [66] is presented in order to attend the issue. The proposed scheme is based on ElGamal signature scheme and provides mutual authentication as well as an increase in performance and resolve on a handful of attacks in relation to previous implementations.

The solution uses two factor authentication and comprehends three operational phases, registration, login and authentication. During the registration phase the user provides his identifier as well as the chosen password. The system uses the ElGamal signature scheme to personalize a smart card with secure information to the user. Manifestly the two factor authentication dwells on the smartcard and the password. On the login phase, the user must present the smart card to a card reader along with the identifier and the password. The smart card then computes verification values to be used on the next phase according to a time interval. On the last phase, the authentication is made by verifying the parameters received by the smart card while processing time validations. If the final comparisons of computed values match, then the user is successfully authenticated.

#### 3.2.1.4 Symmetric Authentication

Symmetric authentication was developed relatively early on the computer security subject. It is a simple and easy to setup method of proving one's identity onto a system. Basically, the symmetric authentication uses one key used for both encryption and decryption, hence the symmetric characteristic. In practice, the key represents a *shared secret* between two or more parties that is used to create a private connection for information exchange. The primary defect of the symmetric cryptography mechanism is present exactly on the previous statement, the common secret is shared with multiple entities. An initial solution to this problem stands in assigning multiple shared secret key pairs per pair of devices, although the secret is still needed to be shared with two different entities to accomplish a communication.

Comparing to the public key authentication method, despite the symmetric authentication being lightweight, with less computational overhead, whereas its counterpart uses more memory and keys, it suffers from the key distribution problem. In order to guarantee a common secret between two entities it must not travel the communication channel on clear text. Symmetric key authentication suggests two methods for distributing the secret, either *in-band* or *out-of-band*. In-band means the secret is shared or deployed between parties using a system running a key distribution protocol. Out-of-band methods implicitly acquiesce keys to be shared previously, outside of the communication channel. Symmetric cryptography authentication methods are widely used on RFID [21] and are mostly associated to challenge-response authentication mechanisms (see Section 3.2.1.2).

Nguyen in his paper [40] discusses an application of the identity-based cryptography, Ibcp as a method of authentication between network devices. The new method proposes two functional aspects, the IBC key distribution and the authentication protocol. Concerning the key distribution, a pair-wise shared key is generated by using a public key - the user's identity - and the private key - delivered by a private key generator. The authentication between devices uses the pair-wise key derived from the remote user's identity, and the local user's private key, it does not need the remote user's private key to calculate the pair-wise key. This is done by using the system's global secret to create the local user private key, and afterwards a function that combines the private key with the remote user's identity to create the symmetric key used for subsequent communications. After the previous process is concluded and the participating entities share a common secret, the authentication protocol exchanges challenge-response messages to verify the validity of each device's secret key. Not only, this solution enables for mutual authentication, it also allows for chaining authentication between devices, translating into an efficient and simple to deploy symmetric authentication protocol.

### 3.2.2 Authorization

#### 3.2.2.1 Access Control Lists

Access control lists specify the permissions of users onto a resource or object. This means that each ACL contains entries that define the subjects to access the object and their allowed operations upon the object. For that matter, in a computer filesystem, ACLs are usually described in the form of *"User X can read file Y"* while on computer networks, ACLs tend to be shaped as *"Traffic from host X is allowed onto service Y"*. Entries are therefore logical statements that can be combined to achieve more complex aggregated permissions. Since each entry on an ACL of an object possesses the permissions of a subject, ACLs grow pro-

portionately to each subject on the system reckoning scalability as a possible issue. Although ACLs are relatively easy to construct and deploy, for larger systems with numerous users they become difficult to manage and understand, also sometimes resulting in redundancy or inconsistency.

*SiRiUS*, as defined in [25], is a secure filesystem layered on top of existing filesystems such as NFS and CIFS. It assumes that the network storage is untrusted and provides, not only its own cryptographic access control, but also end-to-end security for remote requests. Regarding the access control, *SiRiUS* appends each file with a permission file that contains the file's ACL. Each entry on the ACL consists on the encryption of the file's encryption key using the public key of the permitted user. *SiRiUS* does not scale well for a very large number of users.

Distributed computing environments idealized some solutions for ACLs weaknesses. Scalability and reliability are major prepositions of DCE that may reduce server bottleneck on a client-server operation model. This paper [10] presents basic knowledge of distributed systems by illustrating an architecture of the functionalities of DCE components. DCE represents a software technology for setting up and managing data exchange in a distributed system. Users are allowed to access data at remote servers unaware of its location. Access control is assured by a security service module that implements authentication, authorization and encryption. Past the authentication using a private key encryption approach, a remote procedure call is checked for the proper user authorization accesses on a local ACL. Encryption can then be applied to the data exchanged on the communication channel for complete privacy.

#### 3.2.2.2 Capability Lists

A capability list opposes the paradigm used for access control lists. While an ACL is related to objects, a capability list identifies the permissions and operations a subject has or can perform. A capability list is often translated into a key or a token given to a specific legitimate user. The key extends the capability of the subject onto the remote system's resources. As such, a capability can be passed to other entities. This presents advantages and disadvantages on the system's security. When concerning delegation, capability sharing might enable some ease and flexibility if there is a chain of trust between the interacting users. Capabilities are protected from modification of accessing rights but in this case, the capability might be shared with unauthorized entities for malicious intents that do not need to manipulate the authorized user's permissions.

Many systems are conferred with access control mechanisms based on capabilities. One interesting proposition is described in this paper [36]. This model, based on a former capability system [26], presents an integrated approach of authentication and access control for IoT devices that protects from man-in-the-middle, replay and DoS attacks. A capability is implemented as a data structure containing the unique device identifier and the access rights. The devices interact by requesting communications with each other using the capabilities. If the capabilities match after verification, the request is acknowledged and a communication initiates.

Capability-based access control can also be applied to distributed systems. This solution [43] extends an architecture of multiple domains that cooperate to build a *publish-subscribe system*. The access control is made both at an intra-domain and inter-domain level. The coordinating domain invites other domains to join the system, acting as a delegation authority to other domains. Access policies for resources are distributed to authorized access control

managers that later propagate to domain clients.

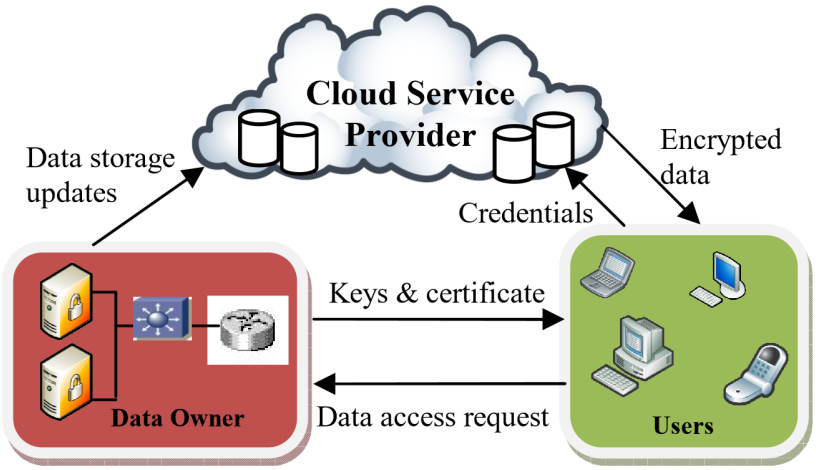



Figure 3.5: Secure access to a cloud service provider. Users request data access and the owner replies with security credentials to be used. The users then access the cloud storage with the same credentials.

In [29], existing solutions for data encryption over cloud computing are boosted through the usage of capability-based access control mechanisms (see Fig. 3.5). The proposed scheme combines access control based on capabilities, with cryptography. The user must first send a register request to the data owner, for accessing the data file with the required access rights. After proper validation and delivery of the accessing credentials to the user, he can request for the actual data existent at the cloud service provider. The data owner sends the file encrypted with credentials, shared only with the user, to the cloud service provider to ensure confidentiality and integrity, along with the capability list. The service provider validates requesting users against the capability list and finally sends them the file.

#### 3.2.2.3 Role-Based Access Control

Restricting access to authorized users through the use of role-based access control, RBAC, is widely accepted as a best practice. Role-based authorization methods stand a reference for worldwide organizations, such as [57] and [52]. This is due to the simplicity of configuration and administration management of roles affected to users and permissions, which are attractive to organizational environments. The properties of RBAC also remain ideal for organizations; the principle of least privilege ensures users are given only the privileges they are required to perform their job, maximizing the roles concept; the separation of duties protects systems from frauds by disallowing a single user having complete control over a transaction; and the delegation of privileges enables role inheritance along a hierarchical chain of command. On organizations, even though permissions associated to roles rarely change, the assignment of roles to users happens frequently.

There are several models defining role-based access control but all of them describe similar basic structural prepositions. Not only there are multiple models, there are also various extensions to primitive models, such as DRBAC [24], TRBAC [9] and LRBAC [50].

A role-based access control model for large-scale web environments was devised in [42]. It presents two different approaches of architectures, called as user-pull and server-pull. Tech-

nologies such as cookies, X.509, SSL and LDAP are integrated with the different architectures in order to provide compatibility with known web technologies. The user-pull architecture centralizes the user as the most active entity where all interactions are initiated from the client to the server.

One of the implementations over a user-pull architecture uses *secure cookies* designed especially for the purpose. On the first step, the user authenticates himself to a role server where he receives his role cookies concealed over encryption. The user then presents the same cookies to the web server, where the correctness and integrity are verified. If everything is successful, the user is given access according to the permissions specified by the web server. In this implementation, RBAC is granted through authentication, confidentiality and integrity of the secure cookies along with asymmetric cryptography.

A more interesting approach serves the server-pull architecture. On this method, the role server is *personified* as an LDAP server where the roles are stored. Oppositely to the previous architecture, on the server-pull the web server is the entity responsible for requesting the authorization permissions of the user. Any requesting subject needs first to authenticate onto the web server. After successful authentication, the server seeks through the LDAP protocol the role of the accessing user. The role is then matched to an entry in the associations list where associations between roles and permissions abide. The web server acknowledges the requesting access by granting the verified permissions to the user that may at this moment access the web server's resources.

The user-pull and the server-pull architectures are compared for their evident properties. Although the performance and the reusability of accessing tokens can be highly appreciated in the user-pull architecture, the server-pull counterpart retains increased user convenience and role freshness, the latter being an important property on the access control subject while still preventing attacks of reusability.

# Chapter 4

# Architecture

According to the stated global objectives, the preeminent plan for the suggested solution is to control access to a remote network host and its services. Obviously, this is a very broad conception. In a more pragmatic view, the overall strategy is to enable the access control at lower layers of the network protocol stack, based on known users' profiles and their accessing security proof. For that objective, the architecture of the project is narrowed down to simpler and modular segments of operations, that specifically intend to resolve each of the small-scope associated obstacles. The architecture is thereby branched into three major operational constituents, illustrated in Fig. 4.1:

- An **access controlling application** that manages all configurations for authorized clients;
- An **access requesting application** that communicates with the access controller to further establish its own security tokens, used for proofing traffic at the remote server;
- A **matching mechanism** that verifies the tokens on received packets, respecting the configurations of the access controlling application.

Since the portrayed scenario prompts for a client-server model, where a client application intends to access a service provided by a remote server host, the access requesting and access controlling applications are respectively associated and instantiated by the client and server hosts. The access controller must guarantee, without any doubt, that the requesting client possesses the proper requirements for accessing the server. For that, the access controller delivers specific *credentials* to the client, so that it can use them to authenticate all of its forthcoming traffic, within a session. The access credentials include a key, an identification number, an algorithm designation, and a role. The identification number is used to establish an association with the user, while the role is used to associate the user with the requested permissions, on a RBAC model. The key and the algorithm are used to apply simple transformations to all the client's outgoing traffic. The transformations are sent in the form of appended **security tokens**, capable of being verified unequivocally of their origin at the server side. From a more specific point of view, there is the need to:

- Create a session with a connecting client, where future traffic is properly controlled of its access into the server;
- Distribute access credentials to the client, in a secure way, to ensure that the traffic belongs to the correct source;

- Admit different user authentication schemes.

Admitting the previous statements as important goals, relevant to provide a legitimate and trustworthy access to network services, it is crucial to establish a secure control channel based on the SSH protocol. SSH is a natural choice as it maps to the stated goals, perfectly matching the architecture. SSH is therefore capable of establishing a secure session with a client, in which the access is limited to the time scope of the session.

Ultimately, a mechanism to verify the security tokens on the server side is needed. Such matching mechanism operates at the server's firewall - the traffic's sole entrance point on the server host, where access control is ideally exercised. The firewall is regulated by the access controller where all credentials and permissions are specified. The matching mechanism uses both the credentials and permissions to authorize ingress traffic from a client host to access the server host, and additionally reach the requested service. As such, there is a clear distinction on **authentication of traffic** entering the server host, and **authorization of traffic** that is later relayed to the appropriate service. Both these concepts will be detailed on subsequent sections.

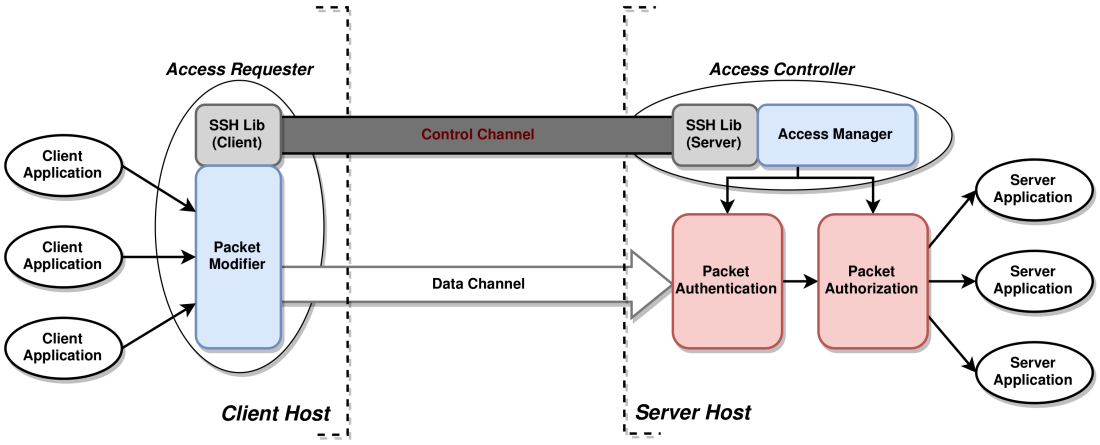



Figure 4.1: Overall system architecture. A control channel is established between the client and server hosts to exchange access credentials. Data packets destined to the server are intercepted and modified by the packet modifier. Such packets are relayed to the server applications if they succeed to pass the authentication and authorization mechanisms. This entire process is only applied on the client→server flow.

Regarding the information coming from client applications, a well-defined path is followed for accommodating a generic traffic flow. An SSH session is started along with the server, and the access credentials are securely exchanged. The entire process for accessing a remote network service is initiated. Whenever a packet destined to the server tries to make its way out of the client's host to the network, it is intercepted by the packet modifier. The packet modifier then uses the session key and the algorithm to build the packet's security token before sending it through the network. The server host inspects the received packet in accordance with the set up credentials and permissions. The packet, if matched to any of the active user↔permission associations, is validated and sent to the target service concluding its operational flow.

On a side note, NAT, despite not being directly associated with the solution at hands, must be considered. This is due to the likeliness of the system being implemented across private and public networks. Therefore, NAT is treated as a component external to the architecture, being merely presented and discussed as an addition to a working proof of concept.

In the following sections, all aforementioned mechanisms will be scrutinized as composite parts of a broad architecture, according to the natural data flow on the system.

## 4.1 Access Information Exchange Through a Control Channel

Authenticated and authorized access to network services is the main concern for the proposed system. By reasoning over this goal, it is natural to consider that information regarding the access control has to be securely exchanged between a service provider and a client. The information has to be used between the participants to apply verifications for controlling access to services. A secure channel must be established in order to guarantee the confidentiality of the exchanged information. Otherwise, any individual could use such crucial knowledge in its own advantage. This architecture relies on the exchange of access information, relevant to the client and the server, through an SSH control channel.

Respecting the usage of SSH, it is possible to relate and identify differences to coexisting solutions such as SSL/TLS and IPsec. SSL/TLS is very often compared to SSH for its remarkably similar concepts on data security, however there are two slight discrepancies that influenced the preference for SSH. SSL/TLS is commonly used to provide security on access to public servers as it uses public-key certificates and public-key infrastructures to validate server hosts and clients. SSH is much more flexible in this matter, where clients users and specific server hosts use pre-shared public keys to establish a secure connection. One other advantage of SSH is that it has a native user authentication layer whereas SSL/TLS simply authenticates two connecting endpoints. The IPsec also suffers from the latter disadvantage while also standing a problem for environments where NAT is deployed.

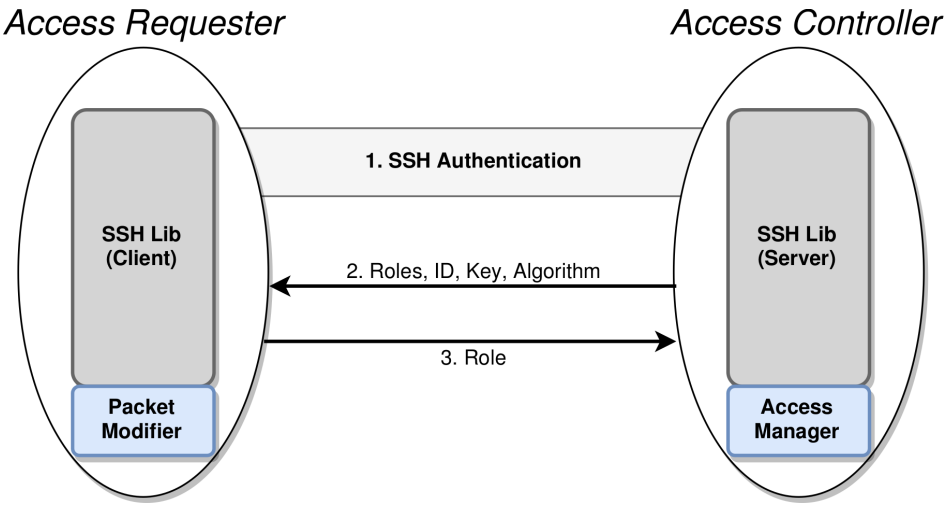



Figure 4.2: Exchanged messages after a successful SSH authentication. The access controlling application sends the credentials to be used by the client and the access requester responds with the desired role that will affect the set of services available during the session.

The information exchanged is essentially constituted by two parts. The first happens after a successful SSH authentication, when the access controller presents the credentials to be used by the access requester. The second part takes place after the first, when the access requester responds to the access controller with the role that will take effect during the accessing session. In the first message, *server→client*, a confidential *session key* is conveyed along with an algorithm designation, to be used by the client for authenticating further packets. Additionally, an identifier is supplied simply to differentiate between clients. Also a list of roles is delivered, representing all the user's permissions for accessing services on the server host. In this step, the **configuration of packets with security tokens** is enabled on the client side. The second message, *client→server*, carries the *role* decision of the accessing user, to clearly distinguish his active access permissions on the server. In this step, the **configuration of the user's accesses to services** is made. After the access controller received the desired role, all its permissions are set as active. All these values are saved on a database cache and deployed into the access controlling mechanisms, recognizing the client as belonging to an active session.

As of this moment, the server is expecting client originated incoming packets to possess correctly configured security tokens, as well as a new layer specification clearly identifying the usage of this process. Using this method, it is possible to verify and ensure that the server is actually communicating with the supposed client. An attacker, having no access to the information exchanged through the control channel, has no way of forging valid packets.

## 4.2 Network-Level User Access Control Protocol

The **Network-level User Access Control** protocol (NUAC) was specially devised for this dissertation in order to aid the solution's implementation. It was conceived considering that there was a need to ensure that the messages exchanged from the client to the server, through the data channel, belong to a legitimate originating entity. Considering that such entity is legitimate, the server perceives it is communicating with an authorized user. Not only it had to withstand both authentication and authorization concepts, but also present a light increase on transmission overhead. This protocol was designed to be similar, yet simpler and more capable, than the AH mode of IPsec. By contrasting with AH, NUAC does not modify the original IP header. Instead, it slightly *adjusts* the payload data. Also, not only it grants packet authentication but also indirect authorization to services. The authorization concept can also be compared to a generic Kerberos system, where users are authorized based on exchanged service tickets and authenticators. However, the usage of Kerberos implies that services must be *Kerberized*, whereas using the the NUAC protocol, this concept is completely transparent to services.

Whenever a client sends packets requesting access to a service, the server must know exactly who is the user responsible for those packets, and whether he is legitimate and authenticated. Only then, the packets are admitted entrance to the server host. The NUAC protocol is used to guarantee exactly that. The access controller informs the access requester that every packet should be appended a **security token** before being sent. Such security token has to be a computed with the provided client-specific security key and the packet data being transmitted.

Observing the underlying transport protocols usually explored by distributed applications, UDP, TCP or another, it is impossible to create a valid and trustworthy notion of *session* that

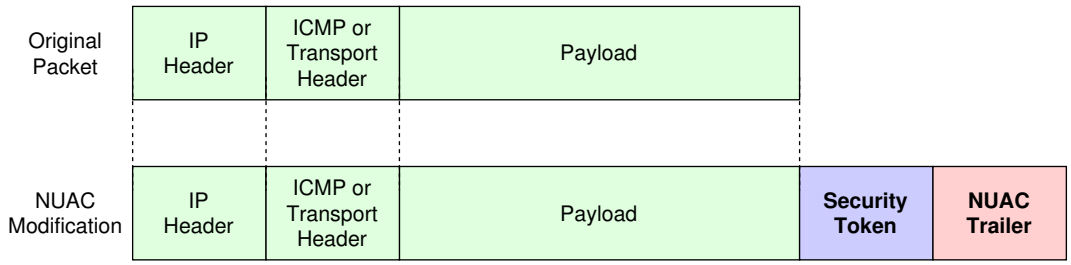



Figure 4.3: NUAC Modification. Both the security token and the NUAC trailer are appended to the original packet during packet modification.

---

would apply to all. In fact, the general idea is to maintain these protocols' modes of operation, not to add a session concept or change an existing one. For that, the security tokens have to be encapsulated in such a way that they do not change how the current TCP/IP stack operates. Also, due to NAT modifications on packet headers' routing information, the security tokens could only be placed inside the packet's payload, without increasing the complexity of including external mechanisms for NAT traversal. The fact that some routers over the Internet remove IP header options [27] also contributed for this decision.

In summary, NUAC plays a role for **authentication at the network level** while being an integrated solution, where coexisting protocols preserve their normal way of functioning, completely unaware of the NUAC protocol.

## 4.3 Calculation of the Security Token Using MAC

The NUAC protocol specification assumes the existence of security tokens imperative to guarantee a successful future packet authentication and authorization. It also determines that those security tokens must be transformations of the client's session key and the packet data, so that the client has a way of legitimizing his packets. A security token is therefore produced by using the session key provided by the server host, the same one it uses for verifying arriving packets, as input of a MAC function.

The decision on the usage of the MAC calculation lies on the advantages over both simple hashes and digital signatures. Hashes per se merely account for integrity checks, meaning that no session key would be used and packets could easily be forged, since there is no shared secret between the genuine producer and the verifier. Digital signatures, besides providing message integrity, also prove authentication and non-repudiation as inherent properties. Digital signatures, however, are more time consuming (both in the generation and in the verification) and take more memory space, thanks to the asymmetric cryptography. Therefore, the decision for using MAC was the most reasonable since it remains as complete, regarding the required objectives of the system, yet simpler, than digital signatures.

Each packet on its outbound course is captured by the packet modifier, where the MAC is calculated. These packets undergo **packet mangling**, consisting in the insertion of a security token calculated by the MAC function, followed by the NUAC trailer. The MAC calculation takes as input the session key and the algorithm, and involves the entire payload that tailgates the network/transport layer, without modifying it. The transformed packets are finally sent to the network for validation at the server-side.

## 4.4 Authenticity and Integrity Verification

The authenticity and integrity verification is applied in this system as a first checkpoint for traffic destined to the server. Both these verifications come together as one since they are intimately related and are performed at the same stage. This checkpoint guarantees that all crossing packets are successfully authenticated and associated to a legitimate originating user.

In order to carry out the verifications, a packet filtering firewall matching mechanism is created and made responsible. Whenever a packet coming from the network reaches the firewall, it passes through a specified set of matching rules. Eventually, on a legitimate packet, a rule will match and the verifications for integrity and authenticity will initiate.

The integrity check is applied simply to detect malformed packets. This means that packets containing wrong NUAC parameters or calculated checksums, or even containing no security tokens, fail to succeed this check. It should be noted that the usual integrity verification of the other protocols is already present and executed before the NUAC integrity check.

The authenticity verification basically checks whether each packet's security token is a correct MAC for the session key and algorithm previously conveyed to the specific originating user. After the successful verification process, the packet is stripped of all NUAC related fields and the existing checksums are recalculated.

Abnormal or incorrect packets that fail either the integrity or the authenticity verification are immediately dropped, thanks to the global strategy regarding the early discard of incorrect packets. Therefore, attackers' packets are discarded right in the beginning of the entire process, in order to prevent against DoS attacks and port scanning mechanisms. Attackers also fail when trying to introduce forged packets on the network since they need to append packets with valid NUAC parameters, using the client's securely exchanged key, in order to create a valid packet.

The mentioned verifications lie as a materialization of the first real barrier for controlling access on every ingress packet. This barrier can simply be interpreted as the **packet authentication**.

## 4.5 Per Service, Role-Based Authorization

One of the overall objectives of the system is to enable an access control to network services, adjustable to each system user. The authorization must then be based on activities the users are entitled to perform onto the system. In role-based access control, the users are precisely associated to their job functions and qualifications. This means that users have a *role* to play when accessing the server host's network services. Roles translate the users' permissions on the system, allowing them to access services according to their role. Role-based authorization accurately represents the hierarchical view of any company or organization according to job positions and inherent responsibilities for accessing resources. This system implements the $\text{RBAC}_0$ model specified in [55]. On a side note, despite the usage of roles having several advantages over concurrent solutions, it can be replaced with other authorization mechanism.

On the system, permissions are defined at the application level and configured beforehand by the server's administrator. Users are also registered on the access controller, identified by a *username*. Having the role's permissions and the users set up, the latter can be assigned one

or several different roles. All the configurations are rendered into a database for persistence and subsequent lookup purposes. The database may also lie as a form of generic component for retrieving access rights, which for instance can be easily switched to an LDAP server.

After a user authenticates himself through SSH, his roles are fetched from the database and are shipped back. The user is then allowed to select one role from the received list. Given the role chosen by the user, the role's associated permissions are retrieved from the database. Such permissions are assembled and activated as packet filtering rules used by the firewall matching mechanism.

At this time, it is in place what is considered to be the second barrier of user-based access control, enabling the server host to enforce on packets the concept of **packet authorization**.

Incoming packets must first pass through the first authentication barrier. If such packets manage to cross it, then they are considered authenticated and are forwarded to the second barrier. On the packet authorization barrier, the permissions of all users with an active session are defined in the firewall. Packets reaching this barrier are distributed to each user's barrier of permissions according to the identification present on the NUAC trailer. The packets follow the firewall's sequential set of rules until they match the intended service. Evidently, the set of rules for each user is distinct unless users are assigned exactly to the same roles, in which case they shall have the same access rights.

Packets effectively enter the system and are sent to the applications only after being dispatched by the firewall. Attackers have no way of passing through packet authentication, therefore there is no real security issue during the authorization process.

# Chapter 5

# Implementation

The system's implementation is the evident translation of the architectural elements into materialized components. In this chapter, a comprehensive description of the system's components is provided. In the first place, essential structures, that are used to facilitate the communication between the components, are defined. The access controller's internal communication is further expanded since it is a quite sophisticated matter. Finally, the actual components' implementation is detailed, dividing itself into four major sections: the kernel module, the `iptables` extension, the access controller application, and the access requester application.

The implementation was accomplished on Linux Kernel 3.19.0 and tested on both kernel 3.19.0 and 4.4.0. The system was delineated to be a standalone package due to its simplicity regarding the deployment and usability.

## 5.1 Structural Specifications

Some structures were defined beforehand since they embrace the vast majority of the solution. The Network-level User Access Control protocol is the primary and most widely present piece, used for authenticating information passing through the components that directly interact with the data channel. In relation to the initial SSH-contained configuration messages, these follow specific structures, that need to be defined for an accurate and appropriate communication. These structures are portrayed in the SSH Exchanged Configuration Messages section. Pertaining to the kernel and user space intercommunication, the *Configfs* Virtual File System is structured as an interface of directories that adopts a pattern relative to the different possibilities of user space applications.

These structures will be rigorously detailed in the following sections.

### 5.1.1 NUAC Protocol

The NUAC protocol is defined as a structure for transmitting **security tokens** between the client and the server. It is encapsulated inside packets' payloads, trailing their original contents. The security tokens correspond to outputs of MAC implementations that use the unmodified packet payloads as input. The MAC calculation also involves a previously distributed key, which in this thesis' case is exchanged through the SSH configuration messages (see Section 5.1.2). In order to grant MAC the property of being a pseudo-random function,

it is used with the HMAC algorithm. Therefore, not only it stands unforgeable under chosen-message attacks, it also possesses a greater collision resistance. As such, the algorithm to use as the underlying cryptographic hash function is also supplied to the MAC function.

In relation to the defined structure, the NUAC protocol consists of a trailer and a data section. The trailer portion is 4-bytes long and contains four mandatory fields (see Fig. 5.1). The data may have a variable size, up to a maximum amount of 256 bytes. The NUAC format is specified as follows:

**Identification**
[2 bytes] Allows for user identification at the client or server end. Its 16 bits, enable the establishment of 65535 possible concurrent clients interacting with the server. Primarily used for identifying the user associated with the current packet. If this field is set to 0, no client identification is used.

**Algorithm**
[4 bits] Up to 15 different HMAC algorithms may be used. If this field is set to 0, no algorithm is used, therefore no security token should be present.

**Reserved**
[4 bits] For future use. Should be set to 0.

**Length**
[1 byte] Indicates the length of the security token in bytes. Allows for a security token with a maximum size of 256 bytes.

**Security Token**
[0 - 256 bytes] Output of a given MAC function. This field intends to provide security at the network layer. This field is optional and if not set, the Algorithm and Length fields should be set to 0.

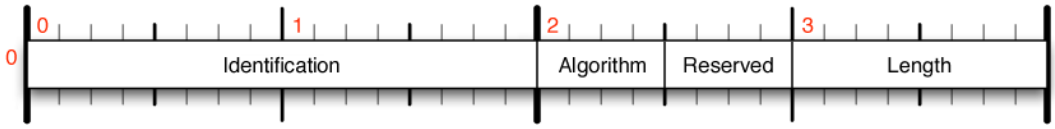



Figure 5.1: NUAC Trailer.

This specification brings about some notes that should be considered. The most evident note is that the data section must precede the NUAC trailer. This is due to the fact that the trailer has to be interpreted first, in order to identify the length of the security token to retrieve from the packet's payload. Secondly, the 4-bit Algorithm field grants the possibility of specifying current well-implemented hash algorithms for the HMAC construction, as well as eventually new algorithms that can output tokens of size up to 2048 bits. On a third note, in the case of different output sizes of the same hash function, only the Length field should be altered while the algorithm stays the same. Lastly, in terms of this thesis, since it stands as a proof of concept, the NUAC protocol was defined simply with three options, out of the complete set[1], regarding the Algorithm field. These options are depicted in Table 5.1.

[1] `https://en.wikipedia.org/wiki/Comparison_of_cryptographic_hash_functions#Parameters`

| Code | Algorithm | Description |
|---|---|---|
| 0 | `NOP` | No Operation |
| 1 | `MD5` | Message Digest 5 |
| 2 | `SHA1` | Secure Hash Algorithm 1 |

Table 5.1: Hash algorithms supported by the NUAC protocol.

### 5.1.2 SSH Exchanged Configuration Messages

The configuration messages exchanged between the client and the server follow specific delineated structures. These configurations are encapsulated in SSH messages in order to experience secure communication through cryptographic encryption. SSH's server and client authentications are extensively described in [65] and [62] respectively, therefore will not be addressed.

Following a successful client↔server authentication, the configuration protocol begins. The first message is originated by the access controlling application. On the overall vision of the configuration protocol, this message represents the **access controller delivering the configurations and requesting the session role**, to be used by the access requester on future service access. The access requesting application responds with a second message represented as the **access requester responding with the session role**. All the exchanged configurations are synthesized in the following list:

- *Identification* - `id` - Associates a client to an active SSH session. Randomly generated integer from a pool of available client IDs. For the first client the pool set is [1, 65535].
- *Algorithm* - `algo` - HMAC algorithm specified by the server administrator.
- *Active Session Key* - `active_sess_key` - Session key to be used for the active SSH session. Created anew every 5 consecutive minutes of an active SSH session.
- *Roles* - `perm_roles` - List of permitted roles for the requesting user. These are established previously on the access controller's database.
- *Active Role* - `active_role` - Role to take effect on the whole duration of the client↔server session.

The designated configurations are stored into a database and are implemented as different types on the system. The identification number and the algorithm are evidently identified by their number representation. The active session key is implemented as a string of 32 random bytes, in its hex representation, suitable for cryptographic use. All the roles are represented as strings and are exchanged as such. The configurations are sent using Python dictionary structures.

The client ID is created whenever a client establishes a session with the server. Once the session is terminated, the client ID number is cleared and made available on the pool of client IDs. The client's ID stays active for the whole duration of the session, even if it passes the 5-minute timeout. The active session key, however, is recreated at the 5-minute mark to guarantee a reasonable freshness of the system's security capability. Also, the decision on the length of the session key is based on the recommendation by NIST to use key lengths greater than 112 bits, when generating or verifying HMACs (chapter 10 in [7]).

### 5.1.3 Configfs Directory Structure

*Configfs* is a component implemented on the kernel as a virtual filesystem. It provides an easily comprehensible interface for the access manager to place its information destined to the firewall module, operating on kernel space. The configuration information is then structured into a directory tree on the virtual filesystem.

In order to be used, *Configfs* must be compiled with the Linux kernel source code. Since it is not mounted by default, it as to be mounted on the `/config` directory to be accessed. The access controller may, at this time, place client's session keys onto the filesystem, according to a defined directory structure. The directory structure is illustrated on Fig. 5.2 as an example.

The root node is represented by the `/config` directory. The kernel module initializes a child node by its name, `xt_hash`. Thereafter each active client gets a directory created by the access controlling application as children of the module directory. The clients' keys are inserted and represented as attributes in the virtual filesystem, respecting each client's folder.

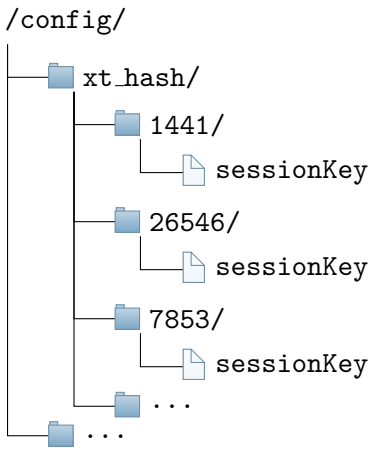



Figure 5.2: One example of the *Configfs* filesystem having three active clients recognized by identification numbers. Inside each client's directory the active session key is stored. The last ellipsis represents external modules possibly using *Configfs*.

## 5.2 Communication Between Kernel and User Space

The defined architecture assumes components to dwell either on the user or kernel space. Due to this approach, the forms of communication between displaced components are not as trivial. In this system, communicating components between the kernel and user space rely on the **transformation** of high-level configurations, existent on the access manager, into low-level ones. This layer conversion is required for the kernel module, as it is the one that uses such configurations to perform calculations, and eventually give access to incoming traffic.

The transformation is unidirectional and necessarily takes on one of two possible paths (see Fig. 5.3). On the first possibility, the configurations are converted into rules with different options, that are placed on the `iptables` extension. The extension then relays the options to the kernel module, internally and transparently. The second possibility relies on the configurations being placed on the virtual filesystem *Configfs*, where the kernel module has direct access. *Configfs* is implemented exclusively on kernel space while providing an interface to

user space applications. Both paths are used by this system for different purposes. However, since the communication inside the `iptables` is transparent for the access manager, only the *Configfs* kernel →user communication is considered.

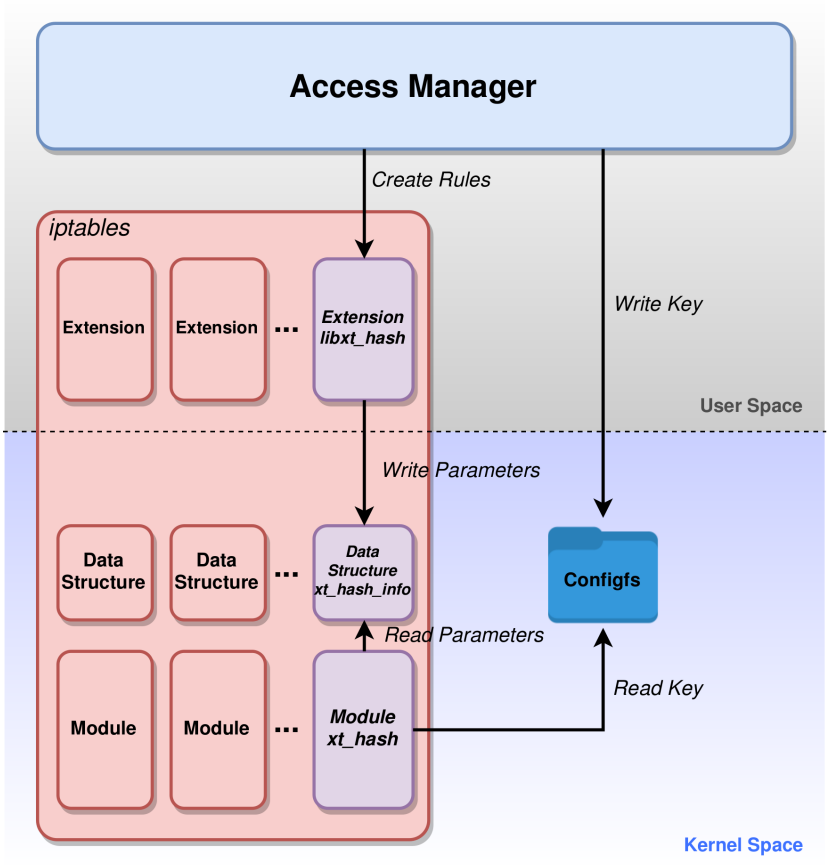



Figure 5.3: Communication between user space and kernel space. The access manager communicates with `iptables` to set up the acess rules. It also deploys the keys directly to the kernel, into the *Configfs* filesystem. The kernel module reads from both sources.

Regarding the virtual filesystem decision, *Configfs* is the only available option for implementing this system's kernel space→user space communication. On both *Procfs* and *Sysfs* the readable and writable attributes are created by the kernel. This means that the access manager could not specify different clients with different algorithms as they connect to the server. The virtual filesystem would be filled with all possible clients even if no client is connected, which becomes completely unnecessary and inefficient for a RAM-based filesystem. On the other hand, the attributes for *Configfs* are created by user space applications as configurations for the kernel. This way, only connected clients are given an entry on the virtual filesystem.

*Configfs* also possesses other advantages on the server's visual communication. By placing key configurations in the virtual filesystem, irrelevant information is diverted to files. Keys do not have to be exposed on the `iptables` listing, rendering it as unreadable for the server administrator. In fact, the keys are available on the database of the access controller and should be consulted in such place.

## 5.3 Component Development

The devised system, in relation to the practical implementation, was designed to comprise four essential components that communicate between one another. Each single component implements different operations according to their main utility for the overall working pipeline. The different elements are written in C or Python code, depending on whether they are implemented at kernel or user space, respectively.

The Kernel Module stands the most fundamental unit and is the only one developed for the kernel. It was designed as such in order to be efficiently fast, and generic in relation to its main purpose. A point of contact between the system's administrator and the kernel module was developed as an `iptables` Extension. This extension operates on user space, directly attached to the `iptables` firewall application, enabling the transmission of optional parameters to kernel space. The Access Controller Application lies in a more central and precursory position, controlling all access to the server. It was developed on user space in order to guarantee an easily understandable and deployable component by the server administrator. Finally, the Access Requester Application's sole purposes are to request access to the remote server and perform packet mangling at the applicational level. It opposes the server-side kernel module that operates on the network layer, since the access requester is intended to be a user-friendly application that is easily used.

### 5.3.1 Kernel Module

The kernel module's primary objective is to verify the authenticity of received packets through a matching mechanism attached to the Linux `iptables` firewall. Several advantages follow through this decision. The `iptables` firewall, for being an application that directly communicates with the Linux kernel, is able to perform the verifications in a completely transparent manner to the user-space access controller application, and inherently to its administrator. For the same reason, the kernel module also permits an increased throughput on matching packets to rules set by the access controller. As such, the kernel module stands a fast mechanism and naturally integrated with `iptables`, as well as all Netfilter[2] matching modules, best suited for this solution.

The kernel module, named `xt_hash`, was outlined as a standalone package, ready to be installed and deployed on any machine with root privileges. Its development was fulfilled holding in credit the Linux Kernel Documentation[3], the Linux Kernel Module Programming Guide[4], and a very complete Netfilter modules guidebook [19].

The matching mechanism is naturally consisted of two working pieces. The first piece implements the user↔kernel communication through the virtual filesystem. The second piece uses the information retrieved from the user space to implement the actual verification of incoming network packets. These two pieces must work together to guarantee the satisfaction of stated objectives. As such, the developed kernel module can be assumed as a concatenation of two cooperative submodules, which will be further addressed as the *Configfs* Subsystem and the `iptables` Kernel Match, respectively.

For deploying the module as a standalone package, the kernel object file `xt_hash.ko` is simply inserted into the kernel. The kernel object file results from the compilation of the

[2] Framework that implements networking utilities on Linux machines.
[3] `https://www.kernel.org/doc/Documentation/`
[4] `http://www.tldp.org/LDP/lkmpg/2.6/html/index.html` by Peter Jay Salzman

kernel module source code.

```
# insmod xt_hash.ko
```

Listing 5.1: Deployment of the module into the kernel by means of the `insmod` utility.

For every module that gets inserted into the kernel, an initializing function is called that must be implemented on the module's source code. Not only the `module_init` function but also a `module_exit` function is necessary. The developed kernel module stands no different and implements the `hash_mt_init` and `hash_mt_exit` as aliases for the previous functions. The initializing function is used to register both the *Configfs* subsystem and the `iptables` kernel match. The cleanup function intends to undo whatever the initializing function did, so it unregisters both submodules.

All the structures and functions related to the kernel module are prototyped in appendix sections B.2 and C.1, respectively.

```
static int __init hash_mt_init(void)
{
    int ret;
    struct configfs_subsystem *subsys;

    subsys = &xt_hash_subsys;
    config_group_init(&subsys->su_group);
    mutex_init(&subsys->su_mutex);
    ret = configfs_register_subsystem(subsys);
    if (ret) {
        printk(KERN_ERR "Error %d while registering subsystem %s\n", ret,
            subsys->su_group.cg_item.ci_namebuf);

        configfs_unregister_subsystem(&xt_hash_subsys);
        return ret;
    }

    printk(KERN_INFO "Hashing module initialized\n");
    return xt_register_match(&xt_hash_mt_reg);
}

static void __exit hash_mt_exit(void)
{
    configfs_unregister_subsystem(&xt_hash_subsys);
    xt_unregister_match(&xt_hash_mt_reg);

    printk(KERN_INFO "Hashing module removed\n");
}

module_init(hash_mt_init);
module_exit(hash_mt_exit);
```

Listing 5.2: Source code of the `hash_mt_init` and `hash_mt_exit`, aliases for `module_init` and `module_exit`, respectively.

### Configfs Subsystem

The *Configfs* subsystem is registered during the initialization of the kernel module. It is possible to identify on Listing 5.2 that the subsystem registration goes through a series of steps. Since the subsystem is an intrinsic part of the module, it is defined as a static `configfs_subsystem` structure, with the module's name. The `configfs_subsystem` structure represents the module in the virtual filesystem as a top-level directory. The subsystem is also a `config_group` and must be initialized as such using the `config_group_init` function. The mutual exclusion call preserves the exclusive association of the defined subsystem to the `xt_hash` module.

In *Configfs* every object is a `config_item`. Items are created and destroyed inside a `config_group`. A group is a collection of items (users in this case) that share the same attributes and operations. Each item possesses a file reflecting a single-value `configfs_attribute`. The implemented subsystem identifies `xt_hash` as both a `configfs_subsystem` and `config_group`, and the different users as `config_item` directories placed inside the group. The attributes are stored as `sessionKey` files, following an overall directory structure similar to the one illustrated in Fig. 5.2.

In order to perform relevant actions onto the virtual filesystem, the operations for items and groups must be established. There are specific structures that define the items' and groups' operations, being the `configfs_item_operations` and `configfs_group_operations`, respectively. The `make_item` operation sets up the possibility to create new items while the `release` operation cleans up after the removal of items. These functions are callbacks called whenever a `mkdir` or `rmdir` action is taken.

For the sake of creating and releasing items, these have to be specifically defined for what they are, and what attributes they retain as well as their possible operations. The actual definition of the items lies on the `simple_child` structure that simply preserves the `sessionKey` attribute. Both the items' attributes and operations reflect the type of items used, and are configured in the `config_item_type` structure.

```
static struct config_item *simple_children_make_item(struct config_group
    *group, const char *name)
{
    struct simple_child *simple_child;

    simple_child = kmalloc(sizeof(struct simple_child), GFP_KERNEL);
    if (!simple_child)
        return NULL;
    memset(simple_child, 0, sizeof(struct simple_child));

    config_item_init_type_name(&simple_child->item, name,
        &simple_child_type);

    simple_child->sessionKey = kzalloc(sizeof(char) *
        XT_HASH_MAX_SESSION_KEY_SIZE + 1, GFP_KERNEL);
    strcpy(simple_child->sessionKey, "0");

    return &simple_child->item;
}
```

Listing 5.3: Source code of the `make_item` function. An item is instantiated with the name (activeID) of the user and is associated to an item type of `simple_child`.

Regarding the items' operations, these translate into simple `show` and `store` callbacks, directly associated to `read` and `write` actions. Due to the fact that the *Configfs* filesystem only recognizes `config` structures, `config_items` must be adapted into `simple_childs` to be manipulated inside the functions via the `to_simple_child` transformation.

### `iptables` Kernel Match

The kernel match is the component that delivers true access control at the network layer. It is also registered during the initialization of the kernel module allowing for the actual matching mechanism to operate, respecting the informations kept by the *Configfs* filesystem. When a packet traverses the `iptables` rules and encounters a rule describing the use of the developed match (`--match hash`), the packet is sent for authenticity verification on the `xt_hash` kernel module. The packet is received on the `hash_mt` function which is entrusted with the accurate assessment of packets. There are yet other functions specified in the `xt_hash_mt_reg` structure considered secondary for the matching purpose.

The `hash_mt` function is an entrance point for the packet's succeeding validation. It is furnished of the packet itself as a socket buffer structure, `skb`, as well as the parameters/options specified by the user-space `iptables` extension as a `xt_action_param` structure.

A socket buffer is a core structure present in the Linux Kernel Networking API in which the kernel handles network packets. A packet received on a network card is implanted into a socket buffer and then passed to the network stack. Packets operations and manipulations, like adding and removing protocol headers, occur by means of the socket buffer structure.

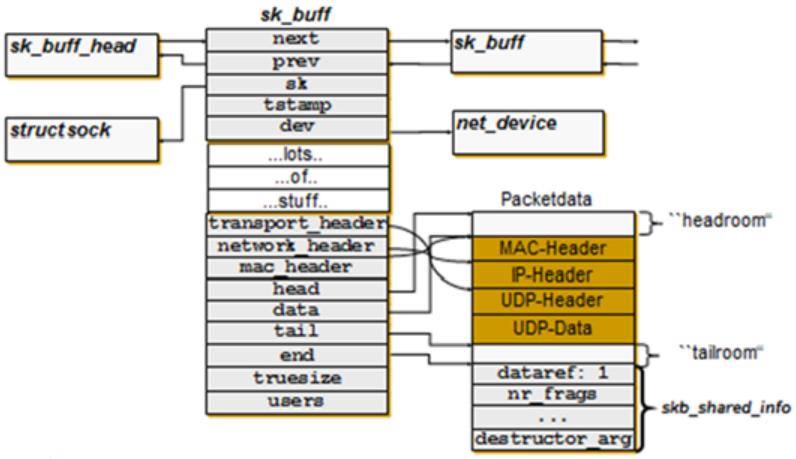



Figure 5.4: Structure of a socket buffer. The packet is enclosed by the data and tail pointers.

The options specified on the `iptables` extension are passed down to the kernel module via a `xt_action_param` structure, which is converted into `xt_hash_info` for being able to be processed the same way it was defined. This is possible thanks to the structure definition file shared between both the `iptables` extension and kernel module, where the first places the rule's information and the second retrieves it for processing.

Having the packet and rule's parameters set, the authenticity verification may be initiated by the worker function `hash_match_it`. In this function, the packet is either ultimately accepted on the current rule's context or is immediately dropped, no packet rejections are

allowed. On Listing 5.4 the example of ICMP validation process is shown.

The `hash_match_it` function starts by decoding the IP header, in which it retrieves the encapsulated protocol, and retains a pointer for the *Configfs* subsystem. According to each protocol the operations performed are slightly different, so a distinction is made on the IP header's identified protocol. A pointer to the initial byte of the NUAC trailer is calculated in order to retrieve NUAC fields and fill the NUAC structure, defined specifically for the kernel module (see appendix section B.1). If such fields represent an invalid NUAC specification the packet is instantly dropped. The packet's payload is retrieved and the MAC is calculated using the `sessionKey` existing at the virtual filesystem. The MAC calculation fails for packets not compliant to the NUAC protocol. The calculated MAC is then compared to the received counterpart and if such comparison fails the packet is also dropped. It should be noted that the calculation of the MAC is made possible through the usage of the Linux Kernel Crypto API. When the authenticity of the packet is finally verified, the NUAC trailer is pulled off and all length and checksum fields are recalculated.

IP packet fragmentation is handled in advance through the linearization of the socket buffer structure. The module is capable of detecting whether a packet is divided into several fragments simply by decoding information on the socket buffer. After the packet is linearized, the validation process becomes a completely transparent operation.

### 5.3.2 `iptables` Extension

The `iptables` extension, named `libxt_hash`, consists on a shared library add-on to `iptables` to include hash validation, a generalized matching on hash security tokens found on incoming NUAC-compliant packets. It directly and transparently interacts with the kernel module to provide the needed tools for a successful verification. Its development is substantially based on the knowledge provided by the Netfilter Documentation[5]. For deploying the extension, the shared library must simply be added to the `xtables` library directory and becomes automatically enabled on the firewall.

```
# cp libxt_hash.so /lib/xtables
```

Listing 5.5: Deployment of the extension into `iptables` as a shared library.

This extension's main purpose is to verify the correctness of the applied matching rules, as well as to produce an interface containing the parameters' information to be utilized by the module. Such interface is defined in the header file of the `xt_hash` module as a `xt_hash_info` structure, and is shown on Listing 5.6. Essentially, the extension operates whenever a rule is included of the hash matching mechanism (`--match hash`). For every specified option in such rule, the `hash_parse` function is called to parse all the options. The options that find themselves to be part of this extension are parsed into the `xt_hash_info` internal structure.

Similarly to every Netfilter extension, the newly developed custom match must be registered within `xtables`. The extension's `_init` function does exactly that and calls for the `xtables_register_match` to register all the operational functions for parsing rules. The `hash_init` function is used to specify default values in case no extra options are inserted. The `hash_parse` follows and parses the extra options, by calling worker parsers to populate the internal communicating structure with the correct parameters. The possible options aggregated to the extension along with their types are identified in the `xt_option_entry`

[5] `https://www.netfilter.org/documentation/`

```
    switch (iph->protocol) {
        case IPPROTO_ICMP:
            icmph = icmp_hdr(skb);
            data_char = skb_tail_pointer(skb) - sizeof(struct nuachdr)

            nuach->id = data_char[0] << 8 | data_char[1];
            nuach->algo = ((data_char[2] << 8 | data_char[3]) & NUAC_ALGO_M)
                >> NUAC_ALGO_LE_OFFSET;
            nuach->reserved = ((data_char[2] << 8 | data_char[3]) &
                NUAC_RESERVED_M) >> NUAC_RESERVED_LE_OFFSET;
            nuach->len = (data_char[2] << 8 | data_char[3]) & NUAC_LEN_M;

            if (nuach->algo != data->algo || nuach->id != data->id ||
                nuach->reserved != 0) {
                ret = NF_DROP;
                break;
            }

            payload_len = skb->len - ip_hdrlen(skb) - sizeof(struct icmphdr)
                - sizeof(struct nuachdr) - nuach->len;
            payload = skb->data + ip_hdrlen(skb) + sizeof(struct icmphdr);

            ret = hmac_vfs_key(hmac_vfs, payload, payload_len, nuach, subsys);
            kfree(payload);
            if (ret != 0)
                break;

            hmac_pkt = data_char - nuach->len;
            ret = memcmp(hmac_pkt, hmac_vfs, nuach->len);
            if (ret != 0)
                break;

            skb_trim(skb, skb->len - sizeof(struct nuachdr) - nuach->len);

            iph->tot_len = htons(skb->len);
            iph->check = 0;
            iph->check = ip_fast_csum(iph, iph->ihl);

            icmph->checksum = 0;
            icmph->checksum = ip_compute_csum((unsigned short *)icmph,
                ntohs(iph->tot_len) - (iph->ihl << 2));

            break;
        ...
    }
```

Listing 5.4: Source code snippet of the ICMP validation process. Three verifications are made to establish the validity of the packet.

```
#ifndef _XT_HASH_H
#define _XT_HASH_H

#include <linux/types.h>

#define XT_HASH_MAX_SESSION_KEY_SIZE 64

...

struct xt_hash_info {
    //__u8 invert;
    __u8 algo;
    __u16 id;
};

#endif /*_XT_HASH_H*/
```

Listing 5.6: Structure shared by the kernel module and the `iptables` extension. Grants an internal and transparent communication channel between the specified rules and their appliance on incoming packets.

structure. There are other secondary functions that merely provide informational outputs mainly destined to the server administrator.

All the structures and functions described in this section are prototyped in sections B.3 and C.2, respectively

### 5.3.3 Access Controller Application

The access controller application is the most dominant and essential component. It acts as a controlling and supervising entity of all the operations that take place on the server side of the system. The application does not deal with any traffic, it exclusively regulates the interactions between each component by communicating control information. Three interactions are considered when managing the system: the communication with the client, rule registration on `iptables`, and configuration of the virtual filesystem.

The communication with the client is guaranteed with the introduction of an SSH server. It is rendered as the implementation of an SSH library's interface onto a Python class. The SSH server reacts to client connection attempts to initiate an SSH session on port `2200`. This is the only port that does not respect authoritative policies due to the fact that, clients must connect themselves and exchange configurations before user authorization can be applied. For that, the port `2200` is the only port that is always open and may suffer from port scanning attacks. Public keys and passwords are the two supported methods for authenticating clients.

Client's configurations are exchanged through the secure SSH channel. When a client requests for a control channel and if SSH's authentication ends up being successful, an SSH session is established with the server host. According to the username received, the access controller follows up with the process of creating a random session key as well as establishing a random and available client id for the session. These configuration parameters, along with the running algorithm set by the server administrator, are sent to the access requester via sockets, encased in a configuration object.

```
# python3 access_controller.py --help
...

optional arguments:
  -c, --create-role ROLE                create a role in database
  -h, --help                            show this help message and exit
  -i, --insert USER                     insert a user in database
  -p, --permissions [PERM [PERM ...]]   premissions for the role
  -r, --roles [ROLE [ROLE ...]]         roles for the user
  ...
  -x, --run ALGO                        run server with algorithm
```

Listing 5.7: Command line syntax of the access controller application. All the specified options relating to users, roles, and permissions can be specified before or during runtime. List, update, and remove options are omitted from this listing.

---

The configurations sent are stored inside a database directly attached to the server application. The database can be interpreted as a dumping ground for caching active sessions' parameters. Not only it accommodates the previous parameters but also the users, their roles, and associated permissions (see Fig. 5.5). As such the database is represented as an abstraction of a permission provider, and it can be easily replaced, with an LDAP server for instance. For accessing the database, the server application requires the use of certain parameters on execution. Listing 5.7 provides an overview of such interactions with the database.

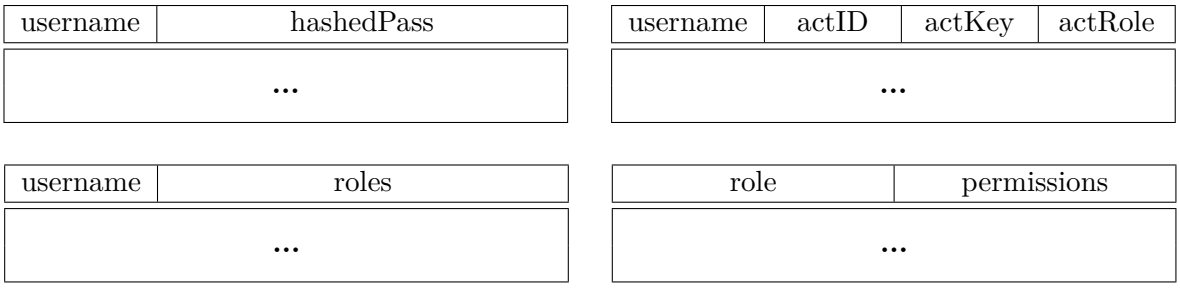



Figure 5.5: Database tables from top left to bottom right: `passwords`, `activeConfigs`, `userConfigs`, `roles`. Each user may have several algorithms and roles, and each role may have several permissions.

Having sent the parameters for the client's configuration, and cached them on a local database, the only action remaining is to enforce the same configurations on the server, to be applied on forthcoming traffic. Some of the configurations sent to the client - the client id and hashing algorithm - are replicated and transformed into `iptables` rules that make use of the new matching module, representing the *authentication barrier*. The user's permissions of the active role are set up on `iptables` chains identified by users' ids, representing the *authorization barrier*. The session key, for being a relatively large string of characters, is placed onto the virtual filesystem where the kernel has direct access.

The three main interactions are therefore epitomized by the `retrieve_configurations`, `deploy_firewall_rules`, and `deploy_keys` functions, respectively. The first function either creates new configurations for a connecting client, or retrieves and recreates them for an

active client that has reached the 5-minute timeout. The second function places the activated client's permissions as rules onto the `iptables`. The last function simply allocates the session keys on the virtual filesystem, to be used by the kernel matching module.

```
    def deploy_keys(username):
        User = Query()
        keys = db.table("keys")
        configs = db.table("configs")

        user = configs.get(User.username == username)
        uid = user["activeID"]

        subprocess.call(["mkdir", "/config/xt_hash/" + str(uid)])

        echo = subprocess.Popen(["echo", keys.get(User.username == \
            username)["activeSessionKey"]], stdout=subprocess.PIPE)
        tee = subprocess.Popen(["tee", "/config/xt_hash/" + str(uid) + \
            "/sessionKey"], stdin=echo.stdout)
        tee.communicate()[0]

        return True
```

Listing 5.8: Function used for deploying keys onto the virtual filesystem. The manipulation of session keys on the VFS is accomplished by the standard `echo` command.

### 5.3.4 Access Requester Application

An access requester application was developed essentially as an SSH library client, to establish a control channel with the remote server. The SSH channel is requested by the client host, to exchange configuration values used on forthcoming traffic. All outgoing traffic is required to be altered according to the NUAC specification, in order to guarantee a successful authentication and authorization of the client's traffic on the server side. The NUAC specification and the functions' prototypes are specified in sections B.1 and C.4, respectively.

After starting the application, a secure communication is initiated between the access requester and access controller. This communication carries the configurations needed for modifying traffic. The access requester expects to receive an id, a hashing algorithm, a session key, and a list of roles. The configurations arrive through sockets binded to the SSH channel, and are stored into Python global objects. After prompting the user for the role to be activated, the exchange is complete and the client assumes that the server host is ready to receive authenticated traffic.

In order to send authenticated traffic, the access requester was extended to support a simple packet mangling mechanism where all packets destined to the server are intercepted and modified. The interception, as shown in Listing 5.9, is made possible through the usage of Netfilter queues (`NFQUEUE`) applied on the output chain of `iptables` by the packet modifier. The SSH traffic, however, must not be intercepted since the SSH channel must be maintained throughout the session and is out of the scope of authorization policies.

Whenever a packet reaches a rule specifying a Netfilter queue, it is sent to the binded callback. The `modify_packet` callback function processes packets according to their protocol, and modifies them by inserting the NUAC trailer along with the security token (see Listing 5.10). The modified packets are then accepted and naturally routed to the server.

```
def nfqueue_start():
    nfqueue = NetfilterQueue()
    nfqueue.bind(1, modify_packet)

    try:
        nfqueue.run()
    except KeyboardInterrupt:
        pass

    print(Packet modification turned off.)
    nfqueue.unbind()
```

Listing 5.9: Function used for sending traffic to the access requesting application. The `NFQUEUE` number 1 is binded between the `iptables` and the application. Packets are sent to the `modify_packet` callback.

```
...
real_payload = bytes(pkt)[((ihl * 4) + (pkt[TCP].dataofs * 4)):]

aalgo = configuration_object["algorithm"]
aid = configuration_object["activeID"]
sess_key = configuration_object["activeSessionKey"]

real_hmac = hmac.new(sess_key.encode(), real_payload, algo)
real_digest = real_hmac.digest()

pkt = pkt / real_digest / HSP(id=aid, algo=aalgo, reserved=0, \
    length=real_hmac.digest_size)

pkt[IP].len = prev_length + 4 + real_hmac.digest_size
del pkt[IP].chksum
del pkt[TCP].chksum
...
```

Listing 5.10: Source code snippet of a TCP packet modification process. A security token is calculated using the payload of the packet. The NUAC trailer and security token are inserted into the packet.

# Chapter 6

# Evaluation

In favor of evaluating the implemented system, this chapter exposes a series of basic tests performed. The initial and most fundamental approach is to verify that the proposed system is working as it is supposed to. This means that the access controller must correctly identify, authenticate, and authorize users requesting for specific services, based on their permissions on the system. For that reason, functionality tests were executed to assert that the traffic originated on a given user is properly verified, and replied if the latter is legitimate.

A following self-evident approach is to acknowledge that the proposed solution is in fact superior than current comparable solutions, according to the initial stated objectives. The proposed solution should be lightweight and efficiently fast when dealing with incoming modified traffic. A set of performance tests were executed, attempting to measure the efficiency of the network traffic flowing between the client and the server, on distinct contexts.

All the executed tests ran on Ubuntu 16.04.1 LTS systems with the Linux Kernel 4.4.0. Due to the fact that the system implementation occurred on the Linux Kernel 3.19.0, some slight negligible alterations were made on the source code in order to comply with the more recent kernel version. The testing environment is additionally detailed in Appendix A.

The sections below illustrate the performed tests categorized as either functional or performance-related. The tests consist essentially on a specified procedure, the expected results, and the actual obtained results. A detailed analysis is presented after each series of tests, to summarize and reason over the obtained results.

## 6.1 Functional Testing

The functional tests intend to guarantee that the implemented system is operating according to the authentication and authorization policies previously specified onto the access manager. The performed tests also intend to prove that, different running protocols, IP fragmentation, and NAT mechanisms, do not disrupt the system and are in fact completely integrated with it.

In order to verify the functionality of the system, a network packet analyzer (Wireshark) was set up in-between the client and the server. The packet analyzer is able to verify network traffic on both flow directions, specially if the modified packets, on the client→server flow, are in accordance with the NUAC specification.

The validation of the system's functionalities is therefore attended by the performed tests, which are summarized in the following items:

1. ICMP check;
2. UDP check;
3. TCP check;
4. GRE check;
5. ICMP check + NAT;
6. UDP check + NAT;
7. TCP check + NAT;
8. GRE check + NAT;
9. ICMP check + IP fragmentation;
10. TCP check + no packet modifier;
11. TCP check + UDP permissions;

The designated tests are simple versions of data exchange between the access requester and the access controller. For the GRE tests, ICMP packets are encapsulated and tunneled through GRE. While the first tests are used for exhibiting successful accesses onto the server, the last two intend to demonstrate that the access control is indeed in place, and no access is granted when certain requirements are not met.

Although the listed tests do not comprise the whole set of possible interactions, they are suitable enough to represent all possible verifications. In light of this fact, the IP fragmentation (Test 9) is only applied on ICMP but stands an equivalent process for the other protocols. Also the TCP checks with no proper access (Tests 10 and 11) represent identical resolutions through ICMP, UDP and GRE. NAT is checked on all protocols since, not only it is applied differently on all used protocols, but it is also considered an important mechanism used on most Internet connected networks.

The tests were accomplished with the usage of the `ping` and `hping3`[1] utilities for their versatility. For the scope of the following tests, a simple nomenclature is used where: $H_{AR}$ stands for the access requester host, and the $H_{AC}$ stands for the access controller host, representing the client and server machines, respectively.

**Test 1** [ICMP check] Protocol validation with permissive policies for ICMP. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpRole*, with ICMP permissions, on $H_{AC}$;
3. Assign the *icmpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;

[1] `https://linux.die.net/man/8/hping3`

6. On $H_{AR}$, send ICMP Echo Request packets to $H_{AC}$.

**Expected Result:** An ICMP Echo Reply packet should arrive at the requesting host for every ICMP Echo Request packet sent.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. ICMP traffic sent afterwards was modified at the client side and validated on the server side. The server replied with regular ICMP Echo Replies.

**Test 2** [UDP check] Protocol validation with permissive policies for UDP port 100. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *udpRole*, with UDP permissions on port 100, on $H_{AC}$;
3. Assign the *udpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send UDP packets with destination port 100 to $H_{AC}$.

**Expected Result:** An ICMP Port Unreachable packet should arrive at the requesting host for every UDP packet sent with destination port 100, since no UDP service is bound to that port.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. UDP traffic sent afterwards was modified at the client side and validated on the server side. The server replied with regular ICMP Port Unreachable packets exposing that no UDP service is bound to port 100.

**Test 3** [TCP check] Protocol validation with permissive policies for TCP port 200. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *tcpRole*, with TCP permissions on port 200, on $H_{AC}$;
3. Assign the *tcpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send TCP packets with destination port 200 to $H_{AC}$.

**Expected Result:** A TCP `RST` packet should arrive at the requesting host for every TCP `SYN` packet sent with destination port 200, since no TCP service is bound to that port.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. TCP traffic sent afterwards was modified at the client side and validated on the server side. The server replied with regular TCP `RST` packets exposing that no TCP service is bound to port 200.

**Test 4** [GRE check] Protocol validation with permissive policies for both GRE and ICMP. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpGreRole*, with ICMP and GRE permissions, on $H_{AC}$;
3. Assign the *icmpGreRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Create a GRE tunnel endpoint on $H_{AC}$;
6. Create a GRE tunnel endpoint on $H_{AR}$;
7. Route all traffic to the $H_{AC}$ through the GRE tunnel, on $H_{AR}$.
8. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
9. On $H_{AR}$, send ICMP Echo Request packets to the remote tunnel endpoint on $H_{AC}$.

**Expected Result:** An ICMP Echo Reply packet should arrive at the requesting host for every GRE-encapsulated ICMP Echo Request packet sent. Traffic sent should be routed via the GRE tunnel.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. GRE traffic sent afterwards was modified at the client side, and received on the tunnel endpoint, de-encapsulated and validated on the server side. The server replied with regular ICMP Echo Replies.

**Test 5** [ICMP check + NAT] Protocol validation with permissive policies for ICMP. NAT gateways are deployed in-between the client and the server.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpRole*, with ICMP permissions, on $H_{AC}$;
3. Assign the *icmpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send ICMP Echo Request packets to $H_{AC}$, passing through both NAT gateways.

**Expected Result:** An ICMP Echo Reply packet should arrive at the requesting host for every ICMP Echo Request packet sent. NAT should not interfere with the proper routing of traffic, nor should it tamper the NUAC trailers or security tokens.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. ICMP traffic sent afterwards was modified at the client side and validated on the server side. The server replied with regular ICMP Echo Replies. The traffic traveling the networks is translated by NAT gateways from a private address into a public address and back.

**Test 6** [UDP check + NAT] Protocol validation with permissive policies for UDP port 100. NAT gateways are deployed in-between the client and the server.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *udpRole*, with UDP permissions on port 100, on $H_{AC}$;
3. Assign the *udpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send UDP packets with destination port 100 to $H_{AC}$, passing through both NAT gateways.

**Expected Result:** An ICMP Port Unreachable packet should arrive at the requesting host for every UDP packet sent with destination port 100, since no UDP service is bound to that port. NAT should not interfere with the proper routing of traffic, nor should it tamper the NUAC trailers or security tokens.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. UDP traffic sent afterwards was modified at the client side and validated on the server side. The server replied with regular ICMP Port Unreachable packets exposing that no UDP service is bound to port 100. The traffic traveling the networks is translated by NAT gateways from a private address into a public address and back.

**Test 7** [TCP check + NAT] Protocol validation with permissive policies for TCP port 200. NAT gateways are deployed in-between the client and the server.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *tcpRole*, with TCP permissions on port 200, on $H_{AC}$;
3. Assign the *tcpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send TCP packets with destination port 200 to $H_{AC}$, passing through both NAT gateways.

**Expected Result:** A TCP `RST` packet should arrive at the requesting host for every TCP `SYN` packet sent with destination port 200, since no TCP service is bound to that port. NAT should not interfere with the proper routing of traffic, nor should it tamper the NUAC trailers or security tokens.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. TCP traffic sent afterwards was modified at the client side and validated on the server side. The server replied with regular TCP `RST` packets exposing that no TCP service is bound to port 200. The traffic traveling the networks is translated by NAT gateways from a private address into a public address and back.

**Test 8** [GRE check + NAT] Protocol validation with permissive policies for both GRE and ICMP. NAT gateways are deployed in-between the client and the server.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpGreRole*, with ICMP and GRE permissions, on $H_{AC}$;
3. Assign the *icmpGreRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Create a GRE tunnel endpoint on $H_{AC}$;
6. Create a GRE tunnel endpoint on $H_{AR}$;
7. Route all traffic to the $H_{AC}$ through the GRE tunnel, on $H_{AR}$.
8. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
9. On $H_{AR}$, send ICMP Echo Request packets to the remote tunnel endpoint on $H_{AC}$, passing through both NAT gateways.

**Expected Result:** An ICMP Echo Reply packet should arrive at the requesting host for every GRE-encapsulated ICMP Echo Request packet sent. Traffic sent should be routed via the GRE tunnel. NAT should not interfere with the proper routing of traffic, nor should it tamper the NUAC trailers or security tokens.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. GRE traffic sent afterwards was modified at the client side, and received on the tunnel endpoint, de-encapsulated and validated on the server side. The server replied with regular ICMP Echo Replies. The traffic traveling the networks is translated by NAT gateways from a private address into a public address and back.

**Test 9** [ICMP check + IP fragmentation] Protocol validation with permissive policies for ICMP. Data size exceeds interface MTU and packet is fragmented. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpRole*, with ICMP permissions, on $H_{AC}$;
3. Assign the *icmpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send ICMP Echo Request packets with data size of 1500 bytes to $H_{AC}$.

**Expected Result:** Two ICMP Echo Reply fragments should arrive at the requesting host for every ICMP Echo Request packet sent with data size of 1500 bytes. This is due to the definition of an MTU of 1500 bytes for the client's interface. The system should rebuild the fragments and accurately identify the NUAC specification.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. ICMP traffic sent afterwards was modified at the client side, and rebuilt and validated on the server side. The server replied with regular fragmented ICMP Echo Replies.

**Test 10** [TCP check + no packet modifier] Protocol validation with no running packet modifier application. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
3. Run an SSH session and login onto the remote server with the correct created credentials, on $H_{AR}$;
4. On $H_{AR}$, send TCP packets with destination port 200 to $H_{AC}$.

**Expected Result:** No packets should arrive at the requesting host for the TCP traffic sent with destination port 200. The `iptables` should filter and discard incoming packets not compliant with the NUAC specification.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. TCP traffic sent afterwards was not modified at the client side and therefore was not validated on the server side. No response is seen from the server.

**Test 11** [TCP check + UDP permissions] Protocol validation with permissive policies for UDP port 200 but not for TCP. No NAT gateways are used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *udpRole*, with UDP permissions on port 200, on $H_{AC}$;
3. Assign the *udpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send TCP packets with destination port 200 to $H_{AC}$.

**Expected Result:** No packets should arrive at the requesting host for the TCP traffic sent with destination port 200. The `iptables` should allow the authentication of packets but not their authorization, since the client is only allowed to access the service bound to UDP port 200.

**Result:** Ok. The requester was successfully authenticated through the correct credentials. TCP traffic sent afterwards was modified at the client side, and authenticated on the server side. The packets were filtered and discarded since the client is not authorized to communicate with TCP port 200. No response is seen from the server.

**Tests Analysis**

Considering the previous tests, it is possible to notice that the procedures for the different protocols are very similar. Also each step produces several automated operations that enable the system to work correctly. In functional terms, the first 8 tests simply create a user on the system, give him the required permissions and validate his packets after running both the access manager and packet modifier applications.

Creating a user and a role, and assigning the latter to the former is made through the access manager application with specific initial parameters (see Listing 5.7). The username and the hashed password, along with the user's supported algorithm, SHA1, are stored in database. Note that, despite the MD5 algorithm is not used on any test, it works identically to SHA1 except that MD5 uses 16-byte tokens whereas SHA1 uses 20-byte's. The permissions are specified for the new role and are also stored. Afterwards, the user is assigned to the created role associated with specific permissions.

Running the access controlling application enables the server to receive incoming SSH connections on port `2200`. When the access requesting application starts on the client side, the SSH credentials are provided and a connection is established, where the configuration values are exchanged. Subsequent traffic is intercepted and modified by the packet modifier that appends the security token (20 bytes) and the NUAC trailer (4 bytes).

A packet received at the server, that unavoidably traverses `iptables` rules, crosses the packet authentication barrier and the packet authorization barrier. For the first 9 tests, the sent packets are correctly generated and are able to bypass both barriers, triggering an automatic system's reply.

As a result, it is viable to infer that each single step described is functioning correctly, due to the fact that all the tests' expected and obtained results match perfectly.

In a more pragmatic view, the first 4 tests acknowledge that the system works properly. The GRE test is executed exclusively using the ICMP protocol, however, encapsulating other protocols is effectively analogous. The four following tests (Tests 5-8) support the evidence that the system is capable of handling network access of clients *hidden* behind NAT devices. This is achievable thanks to the NUAC specification of trailing the fields and the security token on packets.

The last 3 tests are special cases that should be taken into further consideration. They are performed without NAT mechanisms for simplification purposes only, but are obviously transposable and compatible with NAT.

On Test 9, ICMP Echo Requests are sent with data size of 1500 bytes. Since the default MTU value of the Ethernet interface is 1500 bytes, the packets are split into two fragments, one containing 1500 bytes (20 bytes from the IP header + 1480 bytes from the data) and the other containing 72 bytes (20 bytes from the IP header + 8 bytes from the ICMP header + 20 bytes of data + 24 bytes from the NUAC trailer and security token). The obtained results succeed to match the expected, considering that packets are defragmented and handled in their entirety, once inside the kernel module.

Tests 10 and 11 intend to reproduce *modest attacks* by authenticated users. On Test 10, the user simply sends packets to the access control server without having them modified by the packet modifier. The packet authentication barrier inhibits packets not compliant with the NUAC specification to follow through, being therefore discarded. On Test 11, the user attempts to access an unauthorized service port. Despite having its packets authenticated by the NUAC protocol, the user is not authorized to access the service bound to TCP port `200`,

since the only authorization he possesses is to access UDP port `200`. These packets fail to pass the packet authorization barrier and are also discarded.

## 6.2 Performance Testing

The performance tests aim to provide statistical evidence of how efficient is the implemented system, when compared with parallel mechanisms. The tests report two measurements inherently linked to packet processing: the dispatch time within the server, and the network throughput. The first is tested to estimate the time each packet takes to be processed on the `iptables` firewall, added to the reply packet generation. The second measures the rate of successful packet delivery over the network. These tests are therefore capable of contributing with a good performance deduction of the entire system.

Both measurements are evaluated over three different communicational approaches: native transmission, where packets are sent without any alterations; the solution's transmission, where packets are modified according to the NUAC specification; and SSH transmission, where packets are encrypted and tunneled through the network. Each of the measurements is then compared between one another, allowing for a mature and proper tests' analysis.

The performed tests are divided into packet processing and network throughput, respecting both measurements. All the tests are executed within a NAT-enabled environment, however, for simplicity purposes, NAT-related statements are excluded from the tests' descriptions.

#### Server Packet Processing

The following list of items outlines the performed tests on the server's packet processing duration:

1. Native ICMP;
2. Native UDP;
3. Native TCP;
4. Native GRE;
5. Native ICMP + IP fragmentation;
6. Solution ICMP;
7. Solution UDP;
8. Solution TCP;
9. Solution GRE;
10. Solution ICMP + IP fragmentation;
11. SSH ICMP;
12. SSH UDP;

13. SSH TCP;
14. SSH ICMP + IP fragmentation;

The designated tests follow the same principles of the functional tests, meaning that they are also simple versions of data exchange between the access requester and the access controller. Yet, these tests are further extended to transmit 250 packets in 5 spanned series of 50 packets. The minimum, maximum, and average values are calculated and presented further below in the tests' analysis.

The tests were also accomplished with the usage of the `ping` and `hping3` utilities. In order to measure the processing time, the network packet analyzer was placed on the server host, inspecting packets leaving the Ethernet interface. The time in which a reply packet leaves the server is deducted to the time that indicates its request's entrance, resulting in the processing duration inside the server. For the scope of the following tests, a simple nomenclature is used where: $H_{AR}$ stands for the access requester host, and the $H_{AC}$ stands for the access controller host, representing the client and server machines, respectively.

**Test 1** [Native ICMP] Protocol performance testing on packet processing with permissive policies for ICMP. Native transmission is used.

**Procedure:**

1. On $H_{AR}$, send 50 ICMP Echo Request packets to $H_{AC}$;
2. Repeat the above steps 5 times.

**Result:** The average duration of packet processing on the server is 36 µs per packet.

**Test 2** [Native UDP] Protocol performance testing on packet processing with permissive policies for UDP port 100. Native transmission is used.

**Procedure:**

1. On $H_{AR}$, send 50 UDP packets with destination port 100 to $H_{AC}$;
2. Repeat the above steps 5 times.

**Result:** The average duration of packet processing on the server is 42 µs per packet.

**Test 3** [Native TCP] Protocol performance testing on packet processing with permissive policies for TCP port 200. Native transmission is used.

**Procedure:**

1. On $H_{AR}$, send 50 TCP packets with destination port 200 to $H_{AC}$;
2. Repeat the above steps 5 times.

**Result:** The average duration of packet processing on the server is 35 µs per packet.

**Test 4** [Native GRE] Protocol performance testing on packet processing with permissive policies for both GRE and ICMP. Native transmission is used.

**Procedure:**

1. Create a GRE tunnel endpoint on $H_{AC}$;

2. Create a GRE tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the GRE tunnel, on $H_{AR}$.
4. On $H_{AR}$, send 50 ICMP Echo Request packets to the remote tunnel endpoint on $H_{AC}$;
5. Repeat the above steps 5 times.

**Result:** The average duration of packet processing on the server is 46 µs per packet.

**Test 5** [Native ICMP + IP fragmentation] Protocol performance testing on packet processing with permissive policies for ICMP. Data size exceeds interface MTU and packet is fragmented. Native transmission is used.

**Procedure:**

1. On $H_{AR}$, send 50 ICMP Echo Request packets with data size of 1500 bytes to $H_{AC}$;
2. Repeat the above steps 5 times.

**Result:** The average duration of packet processing on the server is 56 µs per packet.

**Test 6** [Solution ICMP] Protocol performance testing on packet processing with permissive policies for ICMP. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpRole*, with ICMP permissions, on $H_{AC}$;
3. Assign the *icmpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send 50 ICMP Echo Request packets to $H_{AC}$;
7. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be slightly higher than the value of the corresponding native test.

**Result:** The average duration of packet processing on the server is 98 µs per packet. It is slower than the corresponding native version.

**Test 7** [Solution UDP] Protocol performance testing on packet processing with permissive policies for UDP. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *udpRole*, with UDP permissions on port 100, on $H_{AC}$;
3. Assign the *udpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;

5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send 50 UDP packets with destination port 100 to $H_{AC}$;
7. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be slightly higher than the value of the corresponding native test.

**Result:** The average duration of packet processing on the server is 111 µs per packet. It is slower than the corresponding native version.

**Test 8** [Solution TCP] Protocol performance testing on packet processing with permissive policies for TCP port 200. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *tcpRole*, with TCP permissions on port 200, on $H_{AC}$;
3. Assign the *tcpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send 50 TCP packets with destination port 200 to $H_{AC}$;
7. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be slightly higher than the value of the corresponding native test.

**Result:** The average duration of packet processing on the server is 95 µs per packet. It is slower than the corresponding native version.

**Test 9** [Solution GRE] Protocol performance testing on packet processing with permissive policies for both GRE and ICMP. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpGreRole*, with ICMP and GRE permissions, on $H_{AC}$;
3. Assign the *icmpGreRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Create a GRE tunnel endpoint on $H_{AC}$;
6. Create a GRE tunnel endpoint on $H_{AR}$;
7. Route all traffic to the $H_{AC}$ through the GRE tunnel, on $H_{AR}$.
8. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
9. On $H_{AR}$, send 50 ICMP Echo Request packets to the remote tunnel endpoint on $H_{AC}$;

10. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be slightly higher than the value of the corresponding native test.

**Result:** The average duration of packet processing on the server is 113 µs per packet. It is slower than the corresponding native version.

**Test 10** [Solution ICMP + IP fragmentation] Protocol performance testing on packet processing with permissive policies for ICMP. Data size exceeds interface MTU and packet is fragmented. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *icmpRole*, with ICMP permissions, on $H_{AC}$;
3. Assign the *icmpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
6. On $H_{AR}$, send 50 ICMP Echo Request packets with data size of 1500 bytes to $H_{AC}$;
7. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be slightly higher than the value of the corresponding native test.

**Result:** The average duration of packet processing on the server is 131 µs per packet. It is slower than the corresponding native version.

**Test 11** [SSH ICMP] Protocol performance testing on packet processing with permissive policies for ICMP. SSH transmission is used.

**Procedure:**

1. Create an SSH tunnel endpoint on $H_{AC}$;
2. Create an SSH tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the SSH tunnel, on $H_{AR}$;
4. On $H_{AR}$, send 50 ICMP Echo Request packets to the remote tunnel endpoint on $H_{AC}$;
5. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be higher than the value of both the corresponding native test and solution's test.

**Result:** The average duration of packet processing on the server is 193 µs per packet. It is slower than both the corresponding native and solution's versions.

**Test 12** [SSH UDP] Protocol performance testing on packet processing with permissive policies for UDP port 100. SSH transmission is used.

**Procedure:**

1. Create an SSH tunnel endpoint on $H_{AC}$;
2. Create an SSH tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the SSH tunnel, on $H_{AR}$;
4. On $H_{AR}$, send 50 UDP packets with destination port 100 to the remote tunnel endpoint on $H_{AC}$;
5. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be higher than the value of both the corresponding native test and solution's test.

**Result:** The average duration of packet processing on the server is 203 µs per packet. It is slower than both the corresponding native and solution's versions.

**Test 13** [SSH TCP] Protocol performance testing on packet processing with permissive policies for TCP port 200. SSH transmission is used.

**Procedure:**

1. Create an SSH tunnel endpoint on $H_{AC}$;
2. Create an SSH tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the SSH tunnel, on $H_{AR}$;
4. On $H_{AR}$, send 50 TCP packets with destination port 200 to the remote tunnel endpoint on $H_{AC}$;
5. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be higher than the value of both the corresponding native test and solution's test.

**Result:** The average duration of packet processing on the server is 188 µs per packet. It is slower than both the corresponding native and solution's versions.

**Test 14** [SSH ICMP + IP fragmentation] Protocol performance testing on packet processing with permissive policies for ICMP. Data size exceeds interface MTU and packet is fragmented. SSH transmission is used.

**Procedure:**

1. Create an SSH tunnel endpoint on $H_{AC}$;
2. Create an SSH tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the SSH tunnel, on $H_{AR}$;
4. On $H_{AR}$, send 50 ICMP Echo Request packets with data size of 1500 bytes to the remote tunnel endpoint on $H_{AC}$;
5. Repeat the above steps 5 times.

**Expected Result:** The value for the duration of a single packet processing should be higher than the value of both the corresponding native test and solution's test.

**Result:** The average duration of packet processing on the server is 432 µs per packet. It is slower than both the corresponding native and solution's versions.

**Network Throughput**

The following list of items summarizes the performed tests on the network throughput:

A. Native UDP;

B. Native TCP;

C. Solution UDP;

D. Solution TCP;

E. SSH UDP;

F. SSH TCP;

The identified tests' procedures are identical to the tests performed over the packet processing duration, with a single exception. In this case, the packets are not produced separately and repeated several times, instead they are generated with random data and transfered in bulk, to measure the network throughput.

The tests were accomplished with the usage of the `iperf`[2] utility. This tool runs with the UDP and TCP protocols since only these stand the actual information carriers meant to transport applicational data. For all UDP tests, an initial bandwidth of 9 Mbit/s is set to transfer 10.7 MB of data. The measured outputs are the real throughput, the time taken to transfer the data, packet jitter, and packet loss. For the TCP tests, the amount of transfered data and network throughput are measured for a specified time frame of 10 seconds. All the tests are repeated 10 times for simple average calculation. For the scope of the following tests, a simple nomenclature is used where: $H_{AR}$ stands for the access requester host, and the $H_{AC}$ stands for the access controller host, representing the client and server machines, respectively.

**Test A** [Native UDP] Protocol performance testing on network throughput with permissive policies for UDP port 100. Native transmission is used.

**Procedure:**

1. Run the `iperf` server on UDP port 100, on $H_{AC}$;
2. On $H_{AR}$, run the `iperf` client and send 10.7 MB of data inside UDP packets with destination port 100 to $H_{AC}$;
3. Repeat the above steps 10 times.

**Result:** The real observable network throughput is in average 9.01 Mbit/s.

**Test B** [Native TCP] Protocol performance testing on network throughput with permissive policies for TCP port 200. Native transmission is used.

**Procedure:**

1. Run the `iperf` server on TCP port 200, on $H_{AC}$;
2. On $H_{AR}$, run the `iperf` client for 10 seconds and send TCP packets with destination port 200 to $H_{AC}$;

[2] `http://manpages.ubuntu.com/manpages/xenial/man1/iperf.1.html`

3. Repeat the above steps 10 times.

**Result:** The real observable network throughput is in average 8.78 Mbit/s.

**Test C** [Solution UDP] Protocol performance testing on network throughput with permissive policies for UDP. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *udpRole*, with UDP permissions on port 100, on $H_{AC}$;
3. Assign the *udpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the `iperf` server on UDP port 100, on $H_{AC}$;
6. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
7. On $H_{AR}$, run the `iperf` client and send 10.7 MB of data inside UDP packets with destination port 100 to $H_{AC}$;
8. Repeat the above steps 10 times.

**Expected Result:** The value for the network throughput should be slightly lower than the value of the corresponding native test.

**Result:** The real observable network throughput is in average 8.97 Mbit/s. It has less throughput than the corresponding native version.

**Test D** [Solution TCP] Protocol performance testing on network throughput with permissive policies for TCP port 200. The solution's transmission is used.

**Procedure:**

1. Create a new user adopting the SHA1 algorithm, on $H_{AC}$;
2. Create a new role, *tcpRole*, with TCP permissions on port 200, on $H_{AC}$;
3. Assign the *tcpRole* to the created user, on $H_{AC}$;
4. Run the access manager application using the SHA1 algorithm, on $H_{AC}$;
5. Run the `iperf` server on TCP port 200, on $H_{AC}$;
6. Run the packet modifier application and provide the correct created credentials, on $H_{AR}$;
7. On $H_{AR}$, run the `iperf` client for 10 seconds and send TCP packets with destination port 200 to $H_{AC}$;
8. Repeat the above steps 10 times.

**Expected Result:** The value for the network throughput should be slightly lower than the value of the corresponding native test.

**Result:** The real observable network throughput is in average 8.75 Mbit/s. It has less throughput than the corresponding native version.

**Test E** [SSH UDP] Protocol performance testing on network throughput with permissive policies for UDP port 100. SSH transmission is used.

**Procedure:**

1. Create an SSH tunnel endpoint on $H_{AC}$;
2. Create an SSH tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the SSH tunnel, on $H_{AR}$;
4. Run the `iperf` server on UDP port 100, on $H_{AC}$;
5. On $H_{AR}$, run the `iperf` client and send 10.7 MB of data inside UDP packets with destination port 100 to $H_{AC}$;
6. Repeat the above steps 10 times.

**Expected Result:** The value for the network throughput should be lower than the value of both the corresponding native test and solution's test.

**Result:** The real observable network throughput is in average 8.38 Mbit/s. It has less throughput than both the corresponding native and solution's versions.

**Test F** [SSH TCP] Protocol performance testing on network throughput with permissive policies for TCP port 200. SSH transmission is used.

**Procedure:**

1. Create an SSH tunnel endpoint on $H_{AC}$;
2. Create an SSH tunnel endpoint on $H_{AR}$;
3. Route all traffic to the $H_{AC}$ through the SSH tunnel, on $H_{AR}$;
4. Run the `iperf` server on TCP port 200, on $H_{AC}$;
5. On $H_{AR}$, run the `iperf` client for 10 seconds and send TCP packets with destination port 200 to $H_{AC}$;
6. Repeat the above steps 10 times.

**Expected Result:** The value for the network throughput should be lower than the value of both the corresponding native test and solution's test.

**Result:** The real observable network throughput is in average 7.68 Mbit/s. It has less throughput than both the corresponding native and solution's versions.

### Tests Analysis

The series of performance tests were executed following a simple and common practice. A baseline performance statistic is provided and the other mechanisms are compared on top of it. The native tests act as the baseline for comparison, therefore, they do not present expected performance results. The implemented solution's tests are evidently compared, along with SSH's extended security mechanism. The comparison of the three transmission mechanisms grants the possibility to establish a ratio between the tested performance and usability, in terms of the required security characteristics.

The native transmissions exhibit natural data exchange between the client and the server, where there is no added external security. The solution's transmission concedes the possibility

of service access control by the cost of a transmission overhead of 24 bytes (20 bytes in the case of MD5) per packet, added to the additional packet processing time within the server. The SSH heavyweight approach, not only changes the properties of the original traffic, but also further increases the expense on both the transmission overhead, mainly due to encryption, and the packet processing to calculate and verify the same encryption.

Tests 1 through 14 collect performance statistics on the packet processing time, while tests A through F achieve network throughput performance values. The packet processing duration is divided into its filtering period and the reply generation period. For this thesis purpose, only the filtering time on `iptables` is taken into consideration, since no changes were made on server's reply generation. The network throughput stands as a more global series of tests that account for the amount of data traveling the network and the time it takes to reach the destination.

Regarding the packet processing tests, it is possible to see that in the native transmissions the processing time is rather similar for every protocol. Minor discrepancies are seen in GRE, since these packets traverse firewall rules two times, and in packet fragmentation, due to the obvious additional computation of fragments.

| | **Native** | **Solution** | **SSH** |
|---|---|---|---|
| **ICMP** | 17/**36**/45 | 51/**98**/113 | 134/**193**/245 |
| **UDP** | 21/**42**/50 | 67/**111**/133 | 142/**203**/258 |
| **TCP** | 18/**35**/43 | 53/**95**/112 | 130/**188**/232 |
| **GRE** | 22/**46**/66 | 68/**113**/126 | - / **-** / - |
| **IP Frag.** | 27/**56**/82 | 82/**131**/178 | 334/**432**/499 |

Table 6.1: Statistical results of the tests performed on packet processing. The portrayed results represent the time a packet spends inside the server host, and take the format Min/**Avg**/Max, in microseconds (µs).

It is expected that the packets modified with the new NUAC protocol take longer to process at the `iptables` firewall. This is mainly due to the verification of the packet's security token against the MAC calculated using the stored session key. The removal of the NUAC trailer and security token virtually takes no time, since the pointer to the end of the data within the socket buffer is simply changed to an earlier address. Also two new checksums are calculated to reassure the packet's validity, the IP checksum, and the encapsulated protocol's checksum, whether ICMP, UDP or TCP. The checksums' calculation also takes a considerable part on the duration of the packet processing. The obtained results show that NUAC-compliant packets take approximately 3 times longer than their native counterparts, for ICMP, UDP and TCP. GRE's disparity is simply due to the firewall double pass. On fragmented packets the security token verification and checksum calculation are performed with longer payloads, nevertheless, they are only made once per packet. For that reason, they are slightly faster than the prevailing 3-times proportion.

The SSH's packet processing is anticipated to be quite slower than both the other methods, considering all the payload has to be decrypted. For the tested packets where there is little

payload, the SSH packet processing is slower than the NAUC method by a *discreet* ratio of 2 times. The most substantial test is when the payload actually contains data, with a size close to the Ethernet MTU. For that, Test 14 shows results close to 4 times the duration of NUAC packet processing and 8 times the duration of the native method. This indicates that the SSH packet processing time depends on the packet's payload size by a high proportionate ratio. For this set of tests, GRE tunneling was not tested since SSH was already deployed as a tunnel.

All the obtained results are detailed in Table 6.1 where it is possible to identify the minimum, maximum and average values for every test. Although the average value is the most significant, the maximum and minimum values intend to demonstrate that each method is possibly faster or slower than the actual average. On packets with more expensive computation methods the deviation is noticeably wider. A more visually appealing and comprehensible chart is shown on Fig. 6.1 that sub-categorizes the different mechanisms within each protocol.

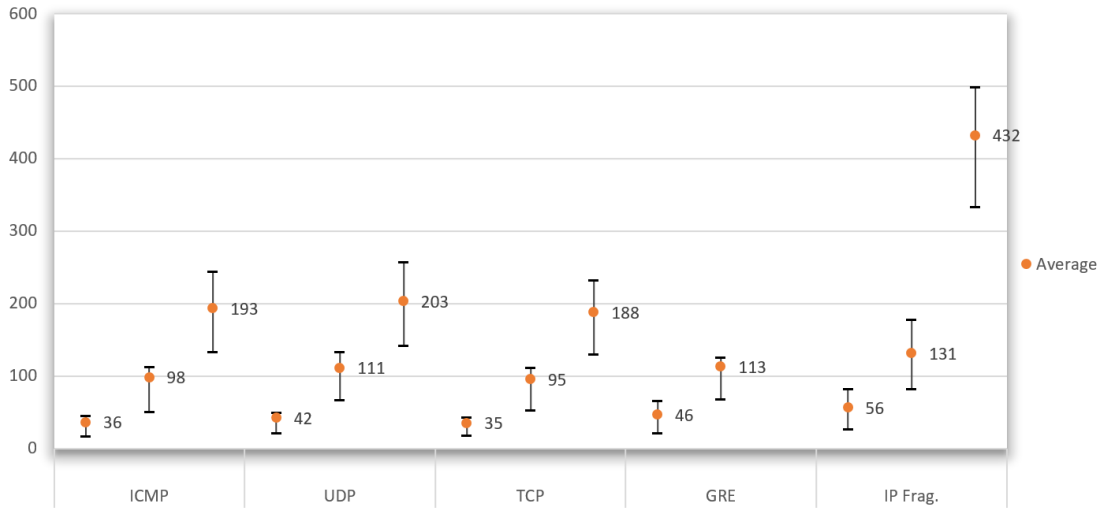



Figure 6.1: Packet processing time on the server host, in microseconds (µs). The results are represented as intervals between the maximum and minimum values. The intervals are divided for the different protocols and further categorized according to the three transmission mechanisms. The average value is highlighted.

Regarding the network throughput, UDP and TCP tests have minor differences. On UDP tests, the data to be transfered and the pretended bandwidth are set as initial parameters. UDP's `iperf` measures the time it took to transfer the data and the real throughput of the network. Packet jitter and packet loss statistics are also provided for a more complete evaluation. On TCP tests, `iperf` simply measures the throughput of the network within the given time interval.

Since the tests were performed through 10 Mbit/s links, an initial bandwidth was set to 9 Mbit/s to guarantee minimum network congestion. Considering UDP's native transmission results, specially the obtained throughput of 9.01 Mbit/s, it is possible to reckon that a higher initial bandwidth would still enable all the data to be transmitted with no packet loss. Not only that, it would also enable a faster transfer. The solution's transmission actually follows similarly behind, however, the packet jitter is increased and the packet loss draws residual values, translating into a slight lesser throughput. The SSH results immediately show that

obviously no packets are lost, since UDP is encapsulated into SSH's TCP. The packet jitter surpasses by 3 times the one identified in the implemented solution. Also the same amount of data takes longer to transfer between the client and the server, resulting in the reduction of the network throughput to 8.38 Mbit/s.

| | **Native** | **Solution** | **SSH** |
|---|---|---|---|
| **Time** (s) | 10.0 | 10.0 | 10.8 |
| **Data** (MB) | 10.7 | 10.7 | 10.7 |
| **Packet Jitter** (µs) | 16±2 | 107±9 | 290±33 |
| **Packet Loss** (%) | 0.00±0.00 | 0.42±0.14 | 0.00±0.00 |
| **Throughput** (Mbit/s) | **9.01±0.00** | **8.97±0.01** | **8.38±0.02** |

Table 6.2: Statistical results of the tests performed on network throughput for UDP. The results are presented as the average of 10 runs with a confidence interval of 95 %.

By analyzing TCP's test results, the native transmission is capable of delivering 10.5 MB of data in 10 seconds translating into an average of 8.78 Mbit/s. Again, the NUAC packets' throughput accompanies very closely its native counterpart, with a value of 8.75 Mbit/s. SSH's transmission rate decreases to 7.68 Mbit/s allowing to transfer only 9.4 MB of data in the same time interval.

| | **Native** | **Solution** | **SSH** |
|---|---|---|---|
| **Time** (s) | 10.0 | 10.0 | 10.0 |
| **Data** (MB) | 10.5±0.03 | 10.5±0.07 | 9.4±0.36 |
| **Throughput** (Mbit/s) | **8.78±0.03** | **8.75±0.06** | **7.68±0.31** |

Table 6.3: Statistical results of the tests performed on network throughput for TCP. The results are presented as the average of 10 runs with a confidence interval of 95 %.

Bearing in mind the `iperf` tests, some deductions can be made. On both UDP and TCP, the throughput is definitely identical. This is due to the fact that the packet modifier application simply appends 24 bytes of the NUAC trailer and security token. The transfer of an overhead of such value is therefore negligible for packets sizes close to the Ethernet MTU. Differently of the solution's transmission, SSH falls short precisely on bigger packets, since the decryption process depends proportionately on the amount of bytes to decrypt. On a side note, the packet jitter values for all UDP transmissions stand insignificant for the deployed network. A NUAC packet loss below 1 % is also insignificant. Both the UDP and TCP tests' results are listed on Tables 6.2 and 6.3, respectively.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

Network services existing throughout the Internet are visible to every connected entity. As such, they stand easily accessible targets to attackers, that may perform security exploits and gain access to unauthorized resources. Although firewalls can impose a first level of security constraints based on accessing hosts, they do not possess the ability to associate client hosts to the actual users operating behind them. Firewalls are therefore incapable of determining whether individual access requests are made by legitimate parties, moreover, if such parties are entitled to access the services from multiple locations across the globe. In order to answer all of these restrictions, a system was conceived and implemented which's primary motivation was to protect network services from unauthorized accesses.

The devised system assimilates and combines high-level accessing rules based on actual users, with low-level operational components where the access control is preferably exercised. As such, the system assumed the firewall to be the sole entrance point of traffic and incorporated it with an improved module that allows the filtering of incoming packets, on different protocols (ICMP, UDP, TCP, and GRE), based on user profiles. In practice, the system consists of a lightweight user authentication and authorization mechanism at the packet level. It allows the negotiation of out-of-band security credentials that are used to elaborate accessing rules on the server host. The rules are enforced distinctively for each user, based on roles and security keys capable of handling current security requirements. The traffic destined to the services is modified to comply with the negotiated credentials.

The system retains the global strategy of discarding incorrect packets early on the operational flow. Therefore, not only it was enriched with new network-level user-based access control, but unauthorized access to services was also prevented immediately at the entrance of the server host. In addition to these two objectives, the system was designed to be completely transparent and user-friendly for both the user and server administrator individuals. Its easily deployable property was achieved thanks to the implementation's intention to provide the system as a standalone package, ready to be installed anywhere.

The implementation prototype, along with the tests performed, has confirmed that such system is not only possible to reproduce and guarantee trustworthy access control, but it is rather efficient when compared to similar solutions. It uses relatively modern concepts, that lack proper documentation, such as the *Configfs* virtual filesystem, and other well-know and widely used concepts such as message authentication codes (MACs) and role-based access

control (RBAC). It also describes new components and protocols that refresh the network security universal subject.

During the performance evaluation phase, the implemented system was compared with the native transmission and SSH's tunneled transmission of packets. Naturally, the native packet transmission had the fastest processing time inside the server as packets are simply accepted once they reach the firewall. The implemented solution followed right after with an increased process time of nearly 3 times the native solution. SSH's transmission was the slowest, being considerably critical on packets with a size of the Ethernet MTU, when compared to the other two types of transmissions.

There are certain drawbacks related to the implemented system that should be noted. First, since the authentication and authorization mechanisms for packets are separated into two barriers, the resulting accessing rules increase in number proportionate to each user's permissions. Second, the time it takes for a packet to be modified on the client side is presumed to be higher than on the server side. Third, the SSH channel port is susceptible to unauthorized access. Fourth, there is no possibility for advanced settings such as network profiles. Lastly, the system currently has no support for version 6 of the IP protocol.

Considering all the above features and drawbacks, it is admissible to find that the proposed objectives were fulfilled. The system is viable to be integrated with any of the identified protocols and can effortlessly be extended to accommodate other protocols. It can be concluded that the access control of users' traffic was in fact enabled at the network level, totally independent of easily forgeable hosts addresses.

## 7.2 Future Work

Having concluded the development of the proposed system, and since it is a new implementation of a proof of concept, it is only reasonable to take notes about future work that would enhance such system with additional features. As such, the future work essentially focuses on the handicaps identified in the previous section.

One identified inconvenience was the increased number of the firewall's accessing rules. Since the authentication barrier of rules is divided from the authorization one, according to each user's permissions, the firewall's listing becomes too populated and almost unreadable to identify possible failures. The solution to this issue would be to merge both packet authentication and authorization barriers. This means that incoming traffic would be accepted into the service in a single step, with one single rule at the firewall. For this to happen, the architecture would have to incorporate keys, not only independent for each client, but at the same time distinct for each service. The kernel module's matching mechanism would then have to validate the security token based on the required service and its associated key.

Despite presumed to be more efficient than the current solution, it also brings other disadvantages. The increased number on firewall rules would convert into an increased number of keys to store both in the server and the client hosts. Also, in environments where NAPT is deployed, the translation information would have to be redundant on the server host, due to the fact that service ports would have to be securely enclosed inside the security token. These two adversities would make the system very unscalable and less lightweight than it currently is.

Another identified issue was the fact that the packet modifier application is believed to take longer to append the security tokens on outgoing packets than their processing time on

the server host. This is most likely due to the packet modifier being on the application level. The solution would comprise the installment of another kernel module on the client host that would insert the security tokens faster than an user-level packet mangler. However, it would be more difficult and not so intuitive for an amateur client user to run the packet modifier module. He would have to run unknown scripts to install the kernel module on his machine.

The server host's SSH port where the control channel resides was recognized to be prone to unauthorized access attempts. There are several mechanisms that can be applied to mitigate this problem, although all of them introduce more complexity to the system. One of the solutions would be to apply port knocking mechanisms before the establishment of the SSH control channel, thus obfuscating the SSH port from attacking parties.

As additional future work, a graphical interface (GUI) could be developed for the access manager application that would be more user-friendly and intuitive for the server administrator. On such application several new features could be implemented as advanced settings, as for instance a network profiler that identifies which type of network it is connected to, and attempts to differentiate different access permissions and security algorithms automatically based on that information. These settings could be maintained in a secluded tab, so that the initial purpose of granting a user-friendly application that can be used by security amateurs would be preserved. The GUI addition was actually projected but due to time constraints was impossible to realize.

The extension to IPv6 of the implemented system is also considered a valid future work development. This extension should be fairly easy to deploy but also requires additional developing time. Having deployed IPv6 on this system, there would be no need to consider NAT environments. As such, the initial solution described in this section, where the packet authentication and authorization barriers are agglutinated into the same matching mechanism, would become more promising and eventually be considered for development.

# Appendices

# Appendix A

# Testing Environment

A detailed overview of the testing environment is described in this appendix. In the following items, the system's hardware and software information, running on both hosts machines, is presented:

- Processor: 8x Intel Core i7-4790 CPU @ 3.60 GHz;
- Memory: 8040 MB;
- Operating System: Ubuntu 16.04.1 LTS;
- Kernel: Linux 4.4.0-43-generic (x86_64);
- C Compiler: GNU C Compiler 5.4.0
- Python: Python 3.5.2

In order to prepare the server host to be able to run the access manager application, some configurations must be made beforehand. Such configurations include the insertion of kernel modules, the mount of the *Configfs* filesystem, and the insertion of the `iptables` extension. On the client host no specific configurations have to be made.

```
/server# cd module
/server/module# make
/server/module# modprobe x_tables
/server/module# modprobe configfs
/server/module# insmod xt_hash.ko
/server/module# mount -t configfs none /config/
/server/module# cd ../extension
/server/extension# make libxt_hash.so
/server/extension# cp libxt_hash.so /lib/xtables/
/server/extension# cd ..
/server/# python3 access_controller.py ...
```

Listing A.1: Script for deploying the access controlling application. The last command runs the application with specified options.

The network architectures and configurations are different according to the types of tests performed. Fig. A.1 illustrates the network architecture for tests that do not use NAT.

Fig. A.2 illustrates the network architecture for NAT-enabled tests. The hub existing between the NAT gateways is used merely to guarantee that the whole system is functioning properly, by placing a network packet analyzer on the hooked host `100.100.100.3`. The hub introduces the constraint of links with a maximum bandwidth of 10 Mbit/s. On the client's NAT gateway, dynamic PAT (or PAT overload) is performed. On the server's NAT gateway, dynamic PAT is performed along with static PAT entries according to the available services at the server host.


Figure A.1: Simple network architecture of the system. The server host runs the access manager application and the client host runs the packet modifier application. No NAT is used.

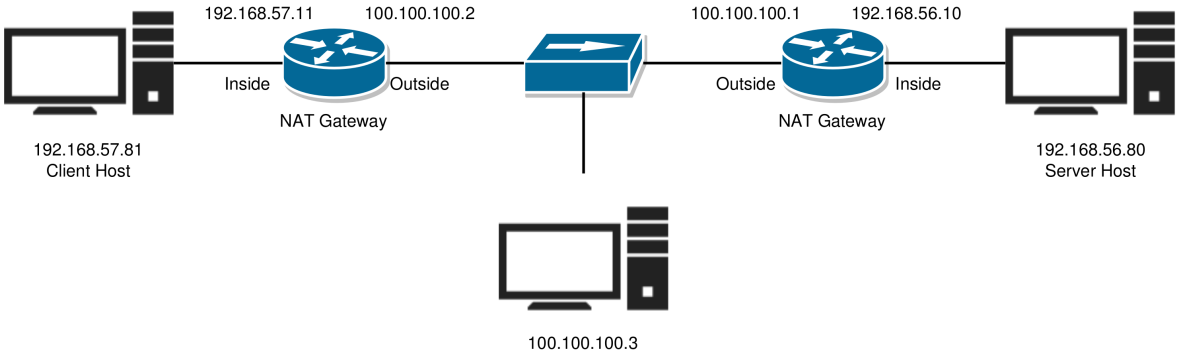



Figure A.2: Network architecture of the system with two NAT Gateways. Both NAT gateways perform NAT/PAT translation. The host directly connected to the hub runs the network packet analyzer.

# Appendix B

# Structures

This appendix intends to provide additional information regarding the structures defined by the different inter-operable components. The information is presented as actual developed source code.

## B.1 NUAC

The NUAC protocol is used by both the client host and server host. It was developed using different programming languages. Therefore, the NUAC protocol is defined for the kernel module as a C structure and for the access requester as a Python3 list descriptor.

```
#ifndef _NUAC_H
#define _NUAC_H

#include <linux/types.h>

#define NOP 0
#define MD5 1
#define SHA1 2

#define MAX_MD5_SIZE 16
#define MAX_SHA1_SIZE 20

struct nuachdr {
    __be16      id;
    __be16      algo:4,
                reserved:4,
                len:8;
};

enum {
    NUAC_ALGO_M             = 0xF000,
    NUAC_RESERVED_M         = 0x0F00,
    NUAC_LEN_M              = 0x00FF
};

enum {
    NUAC_ID_LE_OFFSET       = 16,
    NUAC_ALGO_LE_OFFSET     = 12,
    NUAC_RESERVED_LE_OFFSET = 8,
```

```
    NUAC_LEN_LE_OFFSET      = 0,
    NUAC_ID_BE_OFFSET       = 0,
    NUAC_ALGO_BE_OFFSET     = 16,
    NUAC_RESERVED_BE_OFFSET = 20,
    NUAC_LEN_BE_OFFSET      = 24
};

#endif /*_NUAC_H*/
```

Listing B.1: Source code of the NUAC specification in the kernel module.

```
class NUAC(Packet):
    ...
    fields_desc = [ ShortField("id", None),
                    BitEnumField("algo", 2, 4 {0:"nop",1:"md5",2:"sha1"})
                    BitField("reserved", 0, 4)
                    BitField("length", None, 8) ]
    ...
```

Listing B.2: Source code of the NUAC specification in the packet modifier.

## B.2 Kernel Module

Below, the structures present in the `xt_hash` module can be consulted. The source code is written in the C programming language.

```
/* Structure representing a simple child item (user) */
struct simple_child {
    struct              config_item item;
    char                *sessionKey;
};

/* Structure representing the sessionKey attribute and its permissions */
static struct configfs_attribute simple_child_attr_session_key = {
    .ca_owner           = THIS_MODULE,
    .ca_name            = "sessionKey",
    .ca_mode            = S_IRUGO | S_IWUSR,
};

/* Structure containing all attributes of simple child */
static struct configfs_attribute *simple_child_attrs[] = {
    &simple_child_attr_session_key,
    NULL,
};

/* Structure identifying the simple child operations */
static struct configfs_item_operations simple_child_item_ops = {
    .release            = simple_child_release,
    .show_attribute     = simple_child_attr_show,
    .store_attribute    = simple_child_attr_store,
};

/* Structure representing the type of simple child */
static struct config_item_type simple_child_type = {
    .ct_item_ops        = &simple_child_item_ops,
```

```
    .ct_attrs           = simple_child_attrs,
    .ct_owner           = THIS_MODULE,
};

/* Structure identifying the xt_hash group operations */
static struct configfs_group_operations simple_children_group_ops = {
    .make_item          = simple_children_make_item,
};

/* Structure representing the type of xt_hash group */
static struct config_item_type simple_children_type = {
    .ct_group_ops       = &simple_children_group_ops,
    .ct_owner           = THIS_MODULE,
};

/* Structure representing the subsystem xt_hash */
static struct configfs_subsystem xt_hash_subsys = {
    .su_group = {
        .cg_item = {
            .ci_namebuf = "xt_hash",
            .ci_type = &simple_children_type,
        },
    },
};
```

Listing B.3: Source code of the *Configfs* defined structures.

```
/* Structure representing the kernel match infrastructure */
static struct xt_match xt_hash_mt_reg __read_mostly = {
    .name               = "hash",
    .revision           = 0,
    .family             = NFPROTO_UNSPEC,
    .checkentry         = hash_mt_check,
    .destroy            = hash_mt_destroy,
    .match              = hash_mt,
    .matchsize          = sizeof(struct xt_hash_info),
    .me                 = THIS_MODULE,
};
```

Listing B.4: Source code of the kernel match defined structures.

## B.3 `iptables` Extension

Below, the structures present in the `libxt_hash` extension can be consulted. The source code is written in the C programming language.

```
/* Structure containing all possible options */
static const struct xt_option_entry hash_opts[] = {
    {.name = "id", .id = O_ID, .type = XTTYPE_UINT16, .flags =
        XTOPT_MAND},
    {.name = "sha1", .id = O_SHA1, .type = XTTYPE_NONE, .flags = 0},
    {.name = "md5", .id = O_MD5, .type = XTTYPE_NONE, .flags = 0},
    XTOPT_TABLEEND,
};
```

```
/* Structure representing the iptables match infrastructure */
static struct xtables_match hash_match = {
    .name               = "hash",
    .family             = NFPROTO_UNSPEC,
    .version            = XTABLES_VERSION,
    .size               = XT_ALIGN(sizeof(struct xt_hash_info)),
    .userspacesize      = XT_ALIGN(sizeof(struct xt_hash_info)),
    .help               = hash_help,
    .init               = hash_init,
    .print              = hash_print,
    .save               = hash_save,
    .x6_parse           = hash_parse,
    .x6_options         = hash_opts,
};
```

Listing B.5: Source code of the `iptables` extension defined structures.

# Appendix C

# Function Prototypes

This appendix intends to provide additional information regarding the most important functions defined by the different inter-operable components. Some negligible functions are omitted. The information is presented as actual developed source code.

## C.1 Kernel Module

Below, the functions' prototypes present in the `xt_hash` module can be consulted. The source code is written in the C programming language.

```
/* Function that creates a new item (mkdir) */
static struct config_item *simple_children_make_item(struct config_group
    *group, const char *name);

/* Function that shows a value existing in an attribute*/
static ssize_t simple_child_attr_show(struct config_item *item, struct
    configfs_attribute *attr, char *page);

/* Function that stores a value into an attribute */
static ssize_t simple_child_attr_store(struct config_item *item, struct
    configfs_attribute *attr, const char *page, size_t count);

/* Function that removes an item (rmdir) */
static void simple_child_release(struct config_item *item);

/* Function that transforms a config_item into a simple_child */
static inline struct simple_child *to_simple_child(struct config_item
    *item);
```

Listing C.1: Source code of the *Configfs* related function prototypes.

```
/* Function that serves as entry point the kernel match */
static bool hash_mt(struct sk_buff *skb, struct xt_action_param *par);

/* Function that validates the authenticity of a packet */
static bool hash_match_it(const struct xt_hash_info *data, struct sk_buff
    *skb);

/* Funtion that retrieves the sessionKey from the VFS */
```

```
static int hmac_vfs_key(unsigned char *hmac, unsigned char *text,
    unsigned int text_len, struct hsphdr *hsph, struct configfs_subsystem
    *subsys);

/* Function that calculates an HMAC */
static int calc_hmac(unsigned char *hmac, unsigned char *key, unsigned
    int key_size, unsigned char *text, unsigned int text_size, unsigned
    int algo);

/* Function that destroys dynamically allocated resources */
static void hash_mt_destroy(const struct xt_mtdtor_param *par);

/* Function that checks input parameters */
static int hash_mt_check(const struct xt_mtchk_param *par);
```

Listing C.2: Source code of the kernel match related function prototypes.

## C.2 `iptables` Extension

Below, the functions' prototypes present in the `libxt_hash` extension can be consulted. The source code is written in the C programming language.

```
/* Function that provides default values for the internal structure */
static void hash_init(struct xt_entry_match *m);

/* Function that parses options of a rule */
static void hash_parse(struct xt_option_call *cb);

/* Function that parses the id */
static void parse_id(const char *s, struct xt_hash_info *info);

/* Function that prints --help */
static void hash_help(void);

/* Function that prints the extra match information */
static void hash_print(const void *ip, const struct xt_entry_match
    *match, int numeric);

/* Function that saves the extra match information */
static void hash_save(const void *ip, const struct xt_entry_match *match);
```

Listing C.3: Source code of the `iptables` extension related function prototypes.

## C.3 Access Controller Application

Below, the functions and methods' prototypes present in the access controller application can be consulted. The source code is written in the Python programming language.

```
""" Function that retrieves a client's configurations """
retrieve_configurations(username)

""" Function that deploys a client's permissions into the firewall """
deploy_firewall_rules(username, algo)
```

```
""" Function that deploys a client's session key into the VFS """
deploy_keys(username)

""" Method that verifies the provided credentials - public key """
check_auth_publickey(self, username, key)

""" Method that verifies the provided credentials - password """
check_auth_password(self, username, password)
```

Listing C.4: Source code of the access controller application related function and method prototypes.

## C.4 Access Requester Application

Below, the functions' prototypes present in the access requester application can be consulted. The source code is written in the Python programming language.

```
""" Function that starts an SSH connection """
create_ssh_connection(username, ip)

""" Function that authenticates the client """
ssh_manual_auth(transport, username, ip)

""" Function that checks cached SSH connections """
check_ssh_connections_cache(ip)

""" Function that calculates an HMAC """
calc_hmac(payload)

""" Function that binds the Netfilter queue """
nfqueue_start()

""" Function that modifies a packet """
modify_packet(packet)
```

Listing C.5: Source code of the access requester application related function prototypes.

# Bibliography


[1] H. Al-Bahadili and A. H. Hadi. Network Security Using Hybrid Port Knocking. *IJCSNS*, 10(8):8, 2010.

[2] F. H. M. Ali, R. Yunos, and M. A. M. Alias. Simple Port Knocking Method: Against TCP Replay Attack and Port Scanning. In *Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on*, pages 247–252. IEEE, 2012.

[3] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, IETF, 2009.

[4] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787, IETF, 2007.

[5] J. Aycock, M. Jacobson, et al. Improved Port Knocking with Strong Authentication. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[6] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe. Techniques for Lightweight Concealment and Authentication in IP Networks. *Intel Research Berkeley. July*, 2002.

[7] E. B. Barker and A. L. Roginsky. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. SP-800 131A Revision 1, National Institute of Standards and Technology (NIST), 2015.

[8] J. Becker. configfs, Userspace-Driven Kernel Object Configuration. In `https://www.kernel.org/doc/Documentation/filesystems/configfs/configfs.txt` , Oracle Corporation, 2005.

[9] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-Based Access Control Model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.

[10] M. Bever, K. Geihs, L. Heuser, M. Mühlhäuser, and A. Schill. Distributed Systems, OSF DCE, and Beyond. In *DCEâĂŤThe OSF Distributed Computing Environment Client/Server Model and Beyond*, pages 1–20. Springer, 1993.

[11] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, IETF, 2014.

[12] R. Braden, D. Borman, C. Partridge, and W. W. Plummer. Computing the Internet Checksum. RFC 1071, IETF, 1988.

[13] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker (2nd Edition)*. Addison-Wesley Professional, 2 edition, 2003.

[14] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335, IETF, 2011.

[15] M. De Vivo, E. Carrasco, G. Isern, and G. O. de Vivo. A Review of Port Scanning Techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[16] S. E. Deering and R. M. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, 1998.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, IETF, 2008.

[18] G. Dommety. Key and Sequence Number Extensions to GRE. RFC 2890, IETF, 2000.

[19] J. Engelhardt and N. Bouliane. Writing Netfilter Modules. *Revised, July 3*, 2012.

[20] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, IETF, 2000.

[21] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong Authentication for RFID Systems Using the AES Algorithm. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 357–370. Springer, 2004.

[22] E. B. Fernandez and R. Warrier. Remote Authenticator/Authorizer. *Procs. of PLoP*, 2003.

[23] S. Freire and A. Zúquete. A TCP-layer Name Service for TCP Ports. In *USENIX Annual Technical Conference*, pages 275–280, 2008.

[24] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti. dRBAC: Distributed Role-Based Access Control for Dynamic Coalition Environments. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 411–420. IEEE, 2002.

[25] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *NDSS*, volume 3, pages 131–145, 2003.

[26] L. Gong. A Secure Identity-Based Capability System. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 56–63. IEEE, 1989.

[27] F. Gont, R. Atkinson, and C. Pignataro. Recommendations on Filtering of IPv4 Packets Containing IPv4 Options. RFC 7126, IETF, 2014.

[28] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. RFC 5382, IETF, 2008.

[29] C. Hota, S. Sanka, M. Rajarajan, and S. K. Nair. Capability-Based Cryptographic Data Access Control in Cloud Computing. *International Journal of Advanced Networking and Applications*, 1(01), 2011.

[30] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, IETF, 2014.

[31] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, IETF, 2005.

[32] T. J. Killian. Processes as Files. In *USENIX Summer Conference Proceedings*, pages 203–207, 1984.

[33] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, IETF, 1997.

[34] M. Krzywinski. Port Knocking - Network Authentication Across Closed Ports. *Sys Admin: The Journal for UNIX Systems Administrators*, 12(6):12–17, 2003.

[35] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, IETF, 1996.

[36] P. N. Mahalle, B. Anggorojati, N. R. Prasad, and R. Prasad. Identity Authentication and Capability Based Access Control (IACAC) for the Internet of Things. *Journal of Cyber Security and Mobility*, 1(4):309–348, 2013.

[37] A. I. Manzanares, J. T. Márquez, J. M. Estevez-Tapiador, and J. C. H. Castro. Attacks on Port Knocking Authentication Mechanism. In *International Conference on Computational Science and Its Applications*, pages 1292–1300. Springer, 2005.

[38] C. L. C. Miller. *Next Generation Firewalls for Dummies*. Wiley Publishing Inc., 2011.

[39] P. Mochel. The sysfs filesystem. In *Linux Symposium*, pages 313–326, 2005.

[40] K. V. Nguyen. Simplifying Peer-to-Peer Device Authentication Using Identity-Based Cryptography. In *International conference on Networking and Services (ICNS'06)*, pages 43–43. IEEE, 2006.

[41] R. Oppliger. Internet Security: Firewalls and Beyond. *Communications of the ACM*, 40(5):92–102, 1997.

[42] J. S. Park, R. Sandhu, and G.-J. Ahn. Role-Based Access Control on the Web. *ACM Transactions on Information and System Security (TISSEC)*, 4(1):37–71, 2001.

[43] L. I. Pesonen, D. M. Eyers, and J. Bacon. A Capability-Based Access Control Architecture for Multi-Domain Publish/Subscribe Systems. In *International Symposium on Applications and the Internet (SAINT'06)*, pages 7–pp. IEEE, 2006.

[44] J. Postel. User Datagram Protocol. RFC 768, IETF, 1980.

[45] J. Postel. Internet Control Message Protocol. RFC 792, IETF, 1981.

[46] J. Postel. Internet Protocol. RFC 791, IETF, 1981.

[47] J. Postel. Transmission Control Protocol. RFC 793, IETF, 1981.

[48] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, 2001.

[49] M. Rash. Single Packet Authorization with Fwknop. *login: The USENIX Magazine*, 31(1):63–69, 2006.

[50] I. Ray, M. Kumar, and L. Yu. LRBAC: A Location-Aware Role-Based Access Control Model. In *International Conference on Information Systems Security*, pages 147–161. Springer, 2006.

[51] J. Reynolds. Assigned Numbers: RFC 1700 is Replaced by an On-line Database. RFC 3232, IETF, 2002.

[52] H. Roeckle, G. Schimpf, and R. Weidinger. Process-Oriented Approach for Role-Finding to Implement Role-Based Security Administration in a Large Industrial Organization. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 103–110. ACM, 2000.

[53] P. Samarati and S. C. de Vimercati. Access Control: Policies, Models, and Mechanisms. In *International School on Foundations of Security Analysis and Design*, pages 137–196. Springer, 2000.

[54] R. S. Sandhu. Lattice-Based Access Control Models. *IEEE Computer*, 26(11):9–19, 1993.

[55] R. S. Sandhu, E. J. Coynek, H. L. Feinsteink, and C. E. Youmank. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[56] R. S. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[57] A. Schaad, J. Moffett, and J. Jacob. The Role-Based Access Control System of a European Bank: A Case Study and Discussion. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 3–9. ACM, 2001.

[58] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, IETF, 2001.

[59] P. Srisuresh, B. Ford, S. Sivakumar, and S. Guha. NAT Behavioral Requirements for ICMP. RFC 5508, IETF, 2009.

[60] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, IETF, 1999.

[61] A. Vapen, D. Byers, and N. Shahmehri. 2-clickauth Optical Challenge-Response Authentication. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 79–86. IEEE, 2010.

[62] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, IETF, 2006.

[63] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254, IETF, 2006.

[64] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, IETF, 2006.

[65] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, IETF, 2006.

[66] E.-J. Yoon, E.-K. Ryu, and K.-Y. Yoo. Efficient Remote User Authentication Scheme Based on Generalized ElGamal Signature Scheme. *IEEE Transactions on Consumer Electronics*, 50(2):568–570, 2004.

[67] A. Zúquete. *Segurança em Redes Informáticas*. FCA - Editora de Informática, Lda., 4th edition, 2013.

[68] A. Zúquete, P. Correia, and M. Rocha. A Framework for Enforcing User-Based Authorization Policies on Packet Filter Firewalls. In *IFIP International Conference on Communications and Multimedia Security*, pages 204–206. Springer, 2012.