Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Cybersecurity

Academic Year 2022-2023

Improving Security with PCEPS in PCE-based multi-domain networks

Supervisors

Prof. Paola GROSSO

Prof. Fulvio VALENZA

Ing. Leonardo BOLDRINI

Candidate

Matteo BACHIDDU

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

PCEP
Path Computation Element comunication Protocol

PCEPS
Path Computation Element comunication Protocol Secure

TLS
Transport Layer Security

ABR
Area Border Router

AS
Autonomous System

BGP
Border Gateway Protocol

BGP-LS
Border Gateway Protocol Link-State

CSPF
Constraint based Shortest Path First

FRR
Free Range Routing

GMPLS
Generalized Multi Protocol Label Switching

H-PCE

Hierarchical Path Computation Element

IGP

Interior Gateway Protocol

IPv6

IP version 6

ISIS

Intermediate System to Intermediate System

ISIS-TE

ISIS Traffic Engineering

LDP

Label Distribution Protocol

LSP

Label Switch Protocol

MPLS

Multi Protocol Label Switching

NFV

Network Function Virtualization

OSPF

Open Shortest Path First

OSPF-TE

OSPF Traffic Engineering

PCC

Path Computation Client

PCE

Path Computation Element

RFC

Request For Comments

RSVP

Resource Reservation Protocol

SDN

Software Defined Network

SID

Segment Identifier

SR

Segment Routing

SR-MPLS

MPLS Segment Routing

SR-TE

Segment Routing Traffic Engineering

SRv6

Segment Routing over IPv6 dataplane

TCP

Transmission Control Protocol

TED

Traffic Engineering Database

XML

Extensible Markup Language

MITM

Man-in-the-middle (attack)

Chapter 1

Introduction

Traffic engineering is an important mechanism for Internet network providers that traditionally allows optimal allocation of available network resources to achieve the highest performance in traffic delivery. Routing optimization plays a key role in traffic engineering, finding efficient routes so as to achieve the desired network performance.
Increasing concerns on security of data in transit in the Internet, due to an extensive use of networked communications from end users as well critical service providers, demand a different, more transparent and secure way of using network resources. Traffic engineering is then required to not only search for the highest performance given the available resources, but also having to satisfy security requirements for data in transit by different users. Techniques such as **Constraint-based Shortest Path First**(CSPF) aim at identifying a path that satisfies a set of **quality of service** (QoS) constraints. These techniques are heavily used within the scope of a single domain, where all resources are well known and under the control of a single network operator.
For inter-domain traffic engineering, often **Border Gateway Protocol** (BGP) communities are used. In this case, constraints on QoS are difficult to be satisfied as they require an extended exchange of control messages prior to data-plane communications. Path computation in a large, multi-domain networks like the Internet is complex and may require special computational components and co-operation between the elements in different domains. The **Path Computation Element**(PCE)-based model addresses this problem space in [1].
PCEs can compute paths based on a set of defined constraints and distribute network functionality (e.g. routing) within a network domain. PCEs match the Internet model and coexist with each **Label Switching Router** (LSR) in the network, but are also able to augment functionalities in the network by allowing traffic engineering information to be exchanged between PCEs in different network domains.

There are multiple experiments showing the potential of using the **Path Computation Element Communication Protocol** (PCEP) as a standard communication protocol between PCEs, using well-defined requests and replies, like [2] and [3].
We observed, however, a lack of security features that makes the communication between the PCEs critical. We found potential threats and attack vectors that can be exploited to harm the systems involving PCEP, especially when there are multiple domains involved and the endpoints of the communication are in two different autonomous systems (ASs).
This is where the need for a more secure version of PCEP comes in, to protect the relying parties, for instance against leakage of sensitive data and information or to maintain confidentiality and integrity of the computed paths.
In the following work a new framework will be presented, **UPIN** (User-driven Path verification and control in Inter-domain Networks) [4], which aims to fulfill these new trust requirements for the networks and it is built using the PCE and PCEP technologies. The framework advances the state of the art by defining components needed to incorporate transparency, accountability, and controllability into the network. One way to approach these properties, as stated in [5] is through Responsible Internet, a security paradigm that aims to provide users with more control over the network, both in terms of metadata defining the network's structure and operation as well as how the network transports their data. However nowadays the Internet infrastructure has problems and limitations regarding transparency and control over routing of data. In this scenario, the UPIN framework comes in handy as a framework that supports definition of network behaviour from its users. The user becomes the real driver of how the communication takes place and gains a new awareness of the network that he always lacked until now [6]. User requirements can differ from simple performance indicators (KPIs) to precise and specific functions such as firewalling services or IDS. UPIN consists of a set of functions and components that, when coupled together, enable inter-domain networks to fulfill the requirements of transparency, accountability and controllability. The structure of the framework is composed mainly by four components:

- **Domain Explorer**;
- **Path Controller**;
- **Path Tracer**;
- **Path Verifier**;
- **Frontend**.

To explain briefly the operative flow, once a request for data transfer is issued by the user, the Path controller and the Path Tracer components will guarantee

the transfer occurs according to the request; the Path Verifier component will check if the intent of the user is respected. From the Frontend the user receives a confirmation from the Path Verifier about his intention, more precisely if his intention is respected or not.

1.1 Thesis Scopes

Our research will be focused on one of the main components of UPIN, the Path Controller, which is built upon PCEs devices that rely on the PCEP protocol. It is in charge to set proper forwarding rules based on the preferences chosen by the users and to send them to every routers belonging to the domain.
We will explore the inner protocol insecurities, a way to make a transition to a more secure one defining the new PCEPS protocol and how this one will change the network behaviour in single and multi-domain environment both from a security point of view and a technical one. We will present experimental setups used to test and create the new protocol following also the guidelines of the [7], which devices have been used and the evaluations that we made. In the end we will present the improvements that PCEPS brings and the possible attacks that can still be exploited.
We will answer the question of how we can perform multi-domain segment routing in the Path Controller using a brand new version of the old PCEP protocol and if it is worth building the Path Controller using PCE devices, if this protocol is secure enough for all the system or, on the contrary, it introduces new critical weaknesses and vulnerabilities from a UPIN user point of view that wants to trust this framework.
The rest of the thesis is structured as follows:
First, the concepts of Responsible Internet and the UPIN project, with all its part and functionalities, are explained in chapter 2 to grasp the range of this research; in chapter 3 will be made a deep PCEP protocol analysis, to show how it works, how it is structured, its strong and weak points, with a more focus on the vulnerabilities and possible mitigation; in chapter 4 and 5 the design considerations for the implemented approaches are clarified together with the description of our experimental setups; the implementation of the approaches is further detailed in chapter 6; last, the analysis, experiments and results are provided in chapters 6 and 7 before discussing and making conclusions in chapter 8.

1.2 CompSys 2022

This work was also presented and discussed at the 5th Dutch Computer Science Conference (CompSys 2022), june 8-10 2022, with the extended abstract *Implications of using PCEPS in PCE-based multi-domain networks* [8], a work of Leonardo Boldrini, Matteo Bachiddu and Paola Grosso from University of Amsterdam, MSN department and Politecnico di Torino, Netgroup research group.

Chapter 2

Responsible Internet

Before diving into technical details about the protocols involved and the security issues, it is crucial to talk about Responsible Internet and, after that, present and explain the structure of the UPIN framework, the devices chosen in the architecture, the setups used and the evaluations that we made to build the architecture, in particular regarding the Path Controller, what kind of PCE was used and how the PCE Communication Protocol (PCEP) works.

2.1 Background

The evolution of the Internet in the current days has brought many regions in the world such as Europe, in particular for the policy makers, to be more and more concerned about the concepts of sovereignty and trustworthiness regarding their digital economy, since it often relies on external systems that are located elsewhere. An interesting point is that for instance, the top biggest digital and Internet companies are all from United States or China and none of them are from Europe; moreover the number of European tech companies that are getting acquired from other regions companies is increasing. This development can potentially lead to European service providers and, as a consequence the users, to lose control over data and how they are treated.
To solve this issue, which can be generalized to whatever region, the solutions proposed are mostly from policy makers and consist in new policy proposals leaving almost completely untouched the Internet architecture and infrastructure, without giving them too much attention.
For this reason,following [5], we discuss the notion of Responsible Internet, which proposes a new level of trust and sovereignty for critical service providers and all types of users, building the Internet infrastructure more transparent, accountable and controllable at network level.

Following this approach allows the users to be free from the concept of black box Internet, that is to say the set of all the Internet components that builds the structure of the network and allows the proper functioning of the Internet, but that is almost completely obscure to the users. The notion of Responsible internet brings two new distributed and decentralized systems: the Network Inspection Plane (NIP) and the Network configuration Plane (NCP). They work together with a Policy Framework that apply a set of policies that manage and shape the network.

2.1.1 Network Inspection Plane

The Network Inspection Plane (NIP) uses descriptions of the Internet infrastructure, based on requested measurements, to depict all the chain of operators that manage, process and handle users' data, both in terms of flows and properties. It is really important because it increases transparency and accountability properties. The descriptions bring information about useful properties of network operators, for instance their infrastructure, jurisdiction and which kind of relations they have with other network operators. These descriptions are important because they are information abstracted from the underlying technical infrastructure of network operators, so they can be used also by users that would otherwise not have this kind of knowledge. The NIP also allows to verify the data source used to create the network descriptions, increasing users trust.

2.1.2 Network Control Plane

The NCP permits users to choose how they want the Internet infrastructure to manage their data relying on the NIP information obtained before; in doing so, the controllability is improved significantly. It is a set of control and data plane services that map programmable network functions to users' expectations of the network. The NCP also helps to improve the transparency using open programmable telemetry functions.

2.1.3 Policy Framework

The policy Framework defines a set of policies that network operators have to follow to achieve a proper level of transparency, accountability, controllability and usability. For instance, for a high level responsibility, the policy framework could oblige a network operator to share details regarding its legal jurisdiction, details on its relation with other operators, source code of data and control plane and audits of its software.

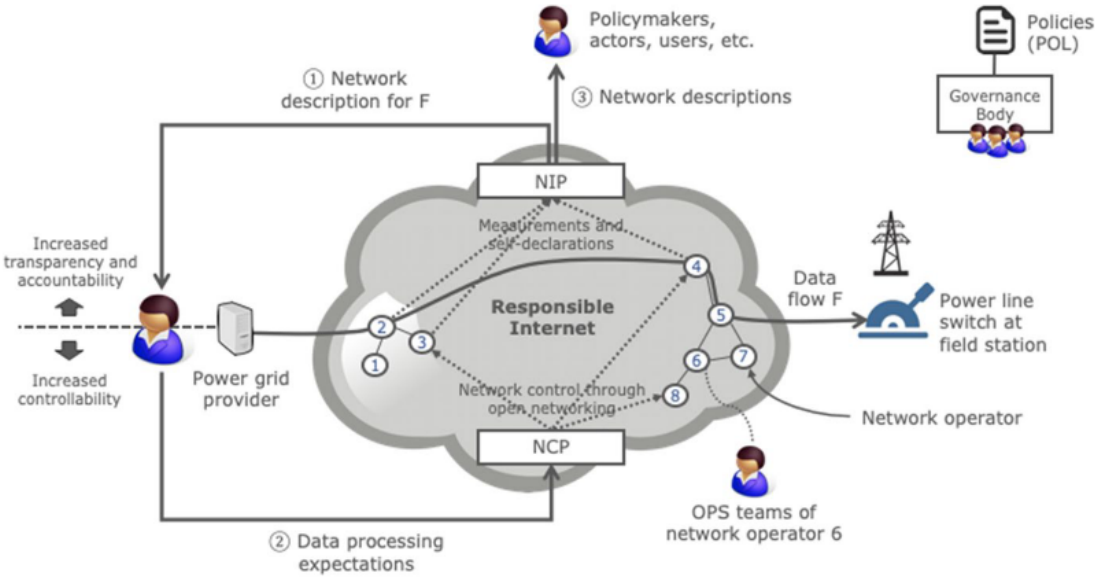

Figure 2.1: Responsible Internet applied in a possible use case scenario with NIP, NCP and a set of policies.

2.1.4 Benefits of the Responsible Internet

Critical infrastructures providers, policy makers and network operators are the ones that benefit the most from Responsible Internet: the first ones gain more control over their dependencies on the network, which is crucial to achieve more security and protection of their services and prevent and mitigate incidents, which otherwise could have a huge impact on the end users; policy makers could use a new type of policy making, using a more data-driven and proactive approach, based on information about the network description but also faster and more efficient policy mediation and policy enforcement; network operators instead can retrieve and use metadata about network descriptions more easily to handle incidents and attacks in a more efficient way.
Furthermore, we expect a wide range of benefits that also the people using the Internet can have from Responsible Internet. For instance, users of VoIP services can check where their data flows end up and how they are handled just easily requesting a network description and, if they want, they can also change the region of the service endpoint.
The UPIN framework, which is the framework used for the work of this thesis, is a possible solution that implements all the components of Responsible Internet and can achieve the transparency, accountability and controllability properties.

2.2 UPIN structure

As we said, the UPIN framework consists in a set of components that, when working together, enable inter-domain networks to achieve the requested requirements of transparency, accountability and controllability. In the table below there is the list of the components and the requirements that each of them allows to achieve:

Table 2.1: Components and fulfilled requirements

Component	Transparency	Accountability	Controllability
Domain Explorer	✓	✓	✓
Path Controller			✓
Path Tracer	✓	✓	
Path Verifier	✓	✓	
Frontend	✓	✓	✓

2.2.1 Domain explorer

The domain explorer is the component responsible to gather, store and keep updated information and metadata about all that is related to security, administrative and environmental properties of the domain's equipment. For instance, typical domain metadata are: routers' source code, organization and composition of the routers in the domain, state of the routers and geographical properties like the location.
This component has a deep and detailed knowledge of its domain, that is information about every node inside the domain. Other domains can request information like metadata about a domain via the domain's frontend which interfaces with the Domain Explorer, for instance to support inter-domain data transfer. Moreover The Domain Explorer also applies policies regarding what type of metadata about the domain wants to share with other domains.

2.2.2 Path Controller

The Path Controller is in charge to set proper forwarding rules based on the preferences chosen by the users and to send them to every routers belonging to the domain.
This component has a local scope, so it has control only over the nodes of its domain and not on the others belonging to another domain.
In figure 2.1 we have two UPIN enabled domains; if the user specifies a particular constraint, the Path Controller must steer the data accordingly.

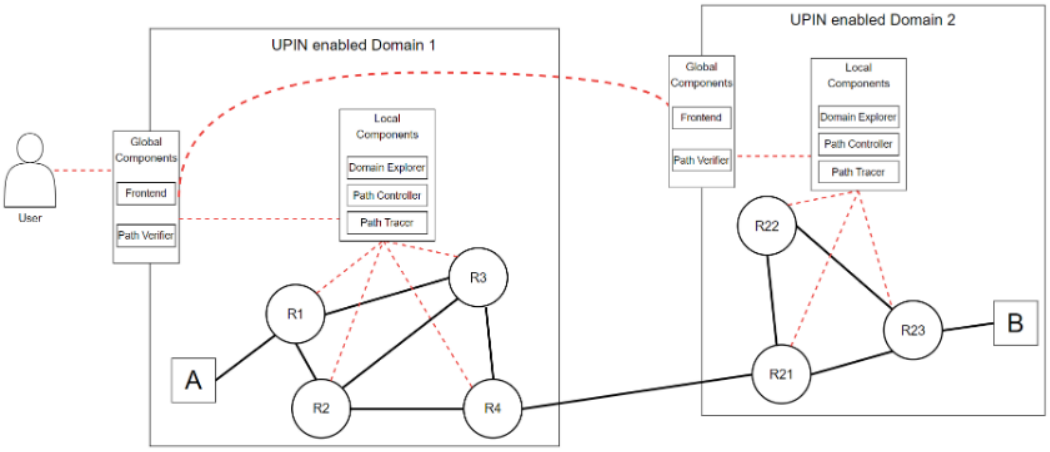

Figure 2.2: UPIN Architecture in 2 UPIN enabled domains.

2.2.3 Path Tracer

This component gathers all the real-time data and information about the traffic in the data plane and stores all the useful information in order to do further verifications.
Tracers are technology indipendent,so it is not specified which kind of implementation should be used per domain, but only the tools that analyze it.

2.2.4 Path Verifier

The Path Verifier is in charge to check if the user's intent is respected or not. In order to do that it uses the original user request and the output data processed by the Path Tracer. The goal of this component is to determine and understand until which level of precision the actual path properties match the user's request. Therefore the verification result is not always certain and well defined, given that traces from the Path Verifier could be incomplete or non-UPIN enabled domains are considered along the transfer of data.
The user can choose different verification grains options, for instance the hop-by-hop verification or domain-by-domain. All it's components, which are running locally in their own domain, gather together all the results of the local verification of other domains.

2.2.5 Frontend

This component allows the communication between user and a UPIN-enabled domain. It is an interface that the user uses to configure the settings. First it

is necessary to set up the destinations and the desired properties to use, then the system computes the available routes that fulfill the requirements using the metadata obtained from the Domain Explorer. With this kind of approach, the user can clearly see if the trust requirements could be met for the destination. Moreover in case the current configuration chosen can't be achieved anymore, the user can configure behaviours to react to this kind of situations.

2.3 VNF chains: Segment Routing & PCEP

It is crucial to discuss also about another concept related to the implementation of the UPIN framework: the Virtual Network Function (VNF) chains using Segment Routing and Path Computation Element Communication Protocol (PCEP) protocol.
Following [2], Network Functions Virtualization (NFV) is an architectural approach that allows to virtualize and deploy dynamically a wide range of network functions, usually hardware-bound, into network nodes to implement communication or architectural services, such as firewall and intrusion detection system. When these virtualized services are all packed and connected together in succession as a multiple of functions, we have the Virtual Network Functions (VNF) chain. The source Routing paradigm, implemented as segment routing (SR), helps the dynamic use of VNFs and VNF chains. With segment routing, each network path is specified at ingress of the network and attached to each packet, so it is possible to direct them to through the proper VNFs.
In the UPIN framework, it is used the SR-MPLS data plane with segment routing for IPv4 together with the PCEP protocol to build and control the paths.

2.3.1 Source Routing & Segment Routing

Source Routing is a routing paradigm that allows to specifies the routes that Internet packets take through the network attaching the complete path inside the header IP . Compared to the standard IP routing, where is the destination that decides the route to follow, Source routing is useful to traffic engineering because allows the users to choose the route to take. This paradigm leads to less overhead for the routers because, as we said before, only the source router has to compute and attach the complete path inside the packets, so all the transit routers only need this information to decide which route to take. On the contrary, the source router has more complexity and the packets have a larger size. On the other hand, from a security point of view, we will explain in later chapters that there is a possibility for Spoofing attack together with a Man in the Middle attack.
This paradigm is implemented in UPIN framework in its variant, the segment routing [9], where the paths are represented as a sequence of segments, each of

them identified by a unique segment identifier (SID). The segments can define a node in the network, a prefix, an adjacency (a link between two nodes) or the anycast traffic.

MPLS Segment Routing

MPLS is a transport protocol that aims to be as simple as possible in order to work theoretically with every other underlying protocol. Each packet has a label, which identifies a path to take and that is used by the transit routers to make the forwarding decisions. Every hop that receives a packet can decide to add a new label, remove the top one or change the current label.
MPLS reduces significantly the complexity and overhead on the routers side compared to the standard IP forwarding.
As stated in [10] and [11], using MPLS inside a network allows to implement segment routing using its variant SR-MPLS, where segments are specified through MPLS labels. So thanks to MPLS it is possible to distribute the segments as labels that have the SID information, using an intra-domain protocol that is to say IGP. Furthermore, this approach gives a global meaning to the labels in the network, as opposed to the normal MPLS that use the labels with a local purpose.
Together with the concept of the Network Function Virtualization, it's easier to define a VNF chain as a sequence of segments, in turn, represented as SR-MPLS labels.

2.3.2 SDN controller & PCEP

For the UPIN framework, the Software Defined Network controller is used, which is useful to support the deployment of the VNF service chains. In our work, the controller needs two functionalities: first, to have a clear picture of the network structure and second to control it. The Border Gateway Protocol (BGP) in its Link State version (BGP-LS) comes in handy to achieve the first functionality, since permits to share, with an external component, traffic information and link state information with a description of links, nodes and prefix in order to have a topological view for the SDN controller.
In order to have control over the network elements, Path Computation Element Communication Protocol is required, which has the goal to communicate and share network paths.

Path Computation Element Communication Protocol

PCEP is a protocol designed and based on Generalized Multi-Protocol Label Switching (GMPLS) paradigm, in order to ease the communication between two new entities explained later in the paragraph, the PCC and PCE. The duty of

PCEP is to create a communication between the entities and using information inside the packets, with SR-LSPs information, help them to compute paths and spread these to the routers in the network.
PCEP defines a specific set of requests and messages:

Table 2.2: PCEP messages

Message Type	**Summary**
Open	First message sent to open a PCEP session
PCReq	This type of message is sent if the PCC requests a path
PCRep	Message sent from the PCE to the PCC as a response
PCUpd	The PCE updates a specific path on the PCC
PCNtf	PCE send a notification to the PCC
PCErr	Message that notifies an error
PCRpt	PCC sends a message to the PCE regarding the path state
Close	Message sent to close the PCEP session

As we said, PCEP introduces two new network entities: the Path Computation Element (PCE), which computes the paths requested and pushes them directly to the Path Computational Client (PCC), the one that can request a path information to be computed.

Path Computation Element

The Path Computation Element (PCE) is an entity inside the network designed to compute paths based on a fixed set of constraints, that is to say the paradigm Constraint-based Shortest Path First (CSPF). It can be stateful or stateless; if it is stateful, the PCE keeps in memory all the paths computed before. A path can be erased from the memory if it is no longer needed or it is destroyed. If the PCE is stateless, it will not keep track of the already computed paths, but uses the Traffic Engineering Database (TED)to compute a new path. The possible workflow between the PCE and PCC is as follow: each PCEP session consists of a Transmission Control Protocol (TCP) connection between two parties. The initial setup of a LSP starts with a PCC sending a path computation request. Then the PCE computes a path and creates a response message that could contain: an error message if was not possible to create the requested path or the information about the path computed. If the PCE wants to update a path, it could send an update message bringing the new label stack for that LSP.

Chapter 3

PCEP Analysis

In this chapter we perform a complete analysis of the protocol, studying his functioning and discussing its packet flows.
Starting from the standardization of PCEP made in [12], we analyse its workflow, some use cases and its insecurities.

3.1 Overview

PCEP is a protocol created with the purpose of allowing the communications between a PCC and a PCE or a PCE and another PCE. Generally, a PCC that ask for a specific LSP sends a path computation request to a PCE using PCEP and in response, the PCE may reply with computed paths that satisfy the constraints given from the PCC.
PCEP works over the Transmission Control Protocol (TCP), so there is no need for PCEP to add new features regarding a reliable connection or control flow.
As in chapter 2, the table below shows the set of PCEP message with a more in depth description of some of them:

- **Open/Keepalive**: the *Open* message is used to initiate a PCEP protocol session with another party; the *Keepalive* message has the purpose to mantain the communication up.

- **PCRep**: this message is sent by a PCE to a PCC as a response of a path computation request and usually contains the LSPs computed, if there were any, or a negative response which specifies the reason of the impossibility to find any path.

- **PCNtf**: the notification can be sent by a PCC to a PCE or by a PCE to a PCC if some event occured.

Table 3.1: PCEP messages

Message Type	Summary
Open	First message sent to open a PCEP session
PCReq	This type of message is sent if the PCC requests a path
PCRep	Message sent from the PCE to the PCC as a response
PCUpd	The PCE updates a specific path on the PCC
PCNtf	A notification sent to notify of a specific element
PCErr	Message that notifies an error
PCRpt	PCC sends a message to the PCE regarding the path state
Close	Message sent to close the PCEP session

A PCC or a PCE can have multiple PCEP sessions with more than one party. There is not a standard mechanism to discover all available PCEs, therefore the set of PCEs can be dynamically discovered or statically configured.

3.2 PCEP packets

PCEP packets have a fixed structure that is composed by a Common Header and a variable length body that depends on the type of message. Some of them have inside the body a mandatory object, which will trigger an error if it is missing, others have an empty body.

3.2.1 Common header

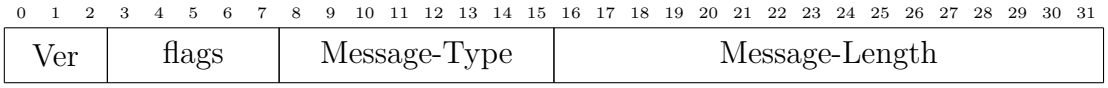

Figure 3.1: Common header

The Common header is composed of four different fields:

- **Version**: this 3-bits field shows the version of the PCEP protocol used;
- **Flags**: 5-bits field that is reserved for future use. Is set to zero on transmission and ignored when the packet is received;
- **Message-Type**: This field (8-bits) shows the type of the PCEP message inside the packet. Its values are from 1 to 7, one number for each type;
- **Message length**: 16-bits field that shows the length in bytes of the PCEP message including the Common header.

3.2.2 Open

It is the first message sent after the establishment of the TCP connection to initiate the PCEP session. It has inside the Common header the type field set to 1. It is composed of the Common header followed by a message body. Inside the body, there are different parameters that are exchanged by the peers and that will be used to set the PCEP session. If both parties agree on this parameters, the new PCEP session can be established, otherwise will be resolved in an error. The packet structure is as follows:

<Open Message>::= <Common Header> <OPEN>

where <OPEN> is the body of the message and consists of one OPEN object and contains the parameters exchanged to set the PCEP session.
When both peers have established the TCP connection, they start the *KeepWait* timer in order to wait for a *Keepalive* message or a *PCErr* message. If the timer expires and none of these messages are received, it will trigger an error to be sent with a *PCErr* message to the other peer and the closing of the TCP connection.
When a peer sends an *Open* message, it starts the *OpenWait* timer in order to wait an *Open* message from the other peer. If this timer expires without having received the *Open* message, it triggers an error to send to the other peer through a *PCErr* message and after that, it will close the TCP connection.

3.2.3 Keepalive

The *Keepalive* is the message sent by both peers to keep the session in the active status. It is also sent after having received the *Open* message to notify it to the other peer. The packet is composed by the Common Header, with the Type field set to 2, followed by an empty body. The packet structure of the *Keepalive* message is as follows:

<Keepalive Message>::= <Common Header> .

As said before, the *Keepalive* message is sent at the frequency specified in the parameter inside the Open message during the initialization of the PCEP session. Sending the *Keepalive* message is optional and depends on the PCE/PCC implementation; also the frequency of the *Keepalive* can be asynchronous from the peers perspective.

3.2.4 PCReq

The *PCReq* is a PCEP message sent by a PCC to a PCE to ask a path computation. Inside this message there can be more than one path computation request. The packet is composed by a Common header, with the Type field set to 3, followed by a body that carries inside the requests.
The structure of the packet is as follows:

- <PCReq Message>::= <Common Header>[<svec-list>]<request-list>;
- <svec-list>::=<SVEC>[<svec-list>];
- <request-list>::=<METRIC>[<metric-list>];
- <request>::=
  <RP><END-POINTS>[<LSPA>][<BANDWIDTH>][<metric-list>]
  [<RRO>[<BANDWIDTH>]][<IRO>][<LOAD-BALANCING>];

where SVEC, RP, END-POINTS, LSPA, BANDWIDTH, METRIC, RRO, IRO and LOAD-BALANCING are all objects.

3.2.5 PCRep

The Path Computation Reply is a message sent by a PCE as a response to a computation request (*PCReq*) received from a PCC. The packet is composed by a Common header, with the Type field set to 4, followed by a body.
The structure of the packet is as follows:

- <PCReq Message>::= <Common Header><response-list>;
- <response-list>::=<response>[<response-list>];
- <response>::=<RP>[<NO-PATH>][<attribute-list>][<path-list>];
- <path-list>::=<path>[<path-list>];
- <path>::=<ERO><attribute-list>;
- <attribute-list>::=[<LSPA>][<BANDWIDTH>][<metric-list>][<IRO>];
- <metric-list>::=<METRIC>[<metric-list>]

It is possible for a PCE to bind many path computation responses into a single *PCRep* if, for instance, a PCC has sent different path computation requests that are received asynchronously by the PCE.
Inside the body there can be a positive or negative response for the related requests. The *PCRep* message contains for each reply inside itself a RP object, which has a Request-Id-number equals to the one carried by the *PCReq* received before.
If the request is positively resolved, the reply contains Explicit Route Objects (EROs) to specify the set of computed paths that match the request. If the request is resolved negatively, the *PCRep* message contains a NO-Path object and information about the reason of the failure.

3.2.6 PCNtf

This message is sent by the PCC or PCE to notify to the other peer that a specific event has occurred. The packet is composed by a Common header, with the Type field set to 5, followed by a body.
The structure of the packet is as follows:

- <PCNtf Message>::= <Common Header><notify-list>;
- <notify-list>::=<notify>[<notify-list>];
- <notify>::=[<request-id-list>]<notification-list>;
- <request-id-list>::=<RP>[<request-id-list>];
- <notification-list>::=<NOTIFICATION><notification-list>.

Inside the *PCNtf* message there is at least one NOTIFICATION object, but there can also be multiple of them if a peer wants to notify the other of different events.

3.2.7 Error message

The *PCErr* message is sent or received when specific conditions that occur trigger an error: a malformed packet is received, a protocol error is met, PCEP parameters specified inside the packets are not allowed or there are policy violations. The packet is composed by a Common header, with the Type field set to 6, followed by a body that carries inside information about the errors.
The structure of the packet is as follows:

- <PCErr Message>::= <Common Header>
  (<error-obj-list> [Open]) | <error>[<error-list>];
- <error-obj-list>::=<PCEP-ERROR>[<error-obj-list>];

- <error>::=[<request-id-list>]<error-obj-list>;
- <request-id-list>::=<RP>[<request-id-list>];
- <error-list>::=<error><error-list>.

The *PCErr* can be sent as a response to a request or in an unsolicited way; if it is sent in response, the *PCErr* message will contain the set of RP objects related to the pending requests that raised the error.

3.2.8 Close

This message is sent by a peer that wants to close the an opened PCEP session. The packet is composed by the Common Header, with the Type field set to 7, followed by a body that carries inside information about the closing procedure. The packet structure of the *Close* message is as follows:

- <Close Message>::= <Common Header><CLOSE> .

When a peer receives the *Close* message, it has to clear all the pending requests, then close the TCP connection and lastly stop the communication with the other peer.

3.2.9 PCEP object format

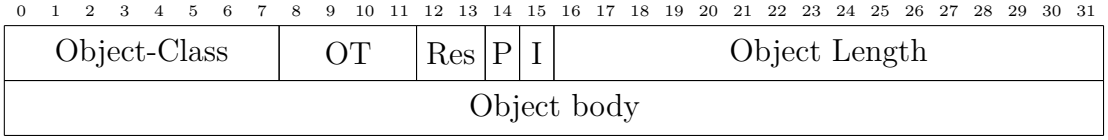

Figure 3.2: Common object header structure

Many types of PCEP messages contain some objects that carry important information. Every object has a structure, precisely they have a fixed common object header followed by a variable body, an object specific field that depends on the type of object. The figure above shows the structure of the common object header, where:

- **Object-class/Object Type(OT)**: identify respectively the PCEP object class and Object type; together they identify uniquely the PCEP object;
- **Res**: flags that identify the reserved fields; they are set to zero by the sender and ignored by the receiver;

- **P**: the processing-rule flag indicates if the object has to be taken into account during the processing of the path computation at the PCE side. It is set by the PCC in a *PCReq* message;
- **I**: the ignore flag is used inside the *PCRep* message and indicates whether an optional object was processed by the PCE or not;
- **Object-length)**: indicates the total length of the object, header included. Object length must be a multiple of 4; moreover the maximum value of the object content length is 65528 bytes.

3.3 Packet Flows

In this section we analyse the packet flows of PCEP protocol going through each phase of a PCEP session, starting from its initialization.

3.3.1 Initialization Phase

Before starting the initialization of a PCEP session comes the establishment of a TCP connection between a PCC and a PCE with the 3-way handshake, only then the initialization phase can start.

After the TCP establishment, the two peers can initiate the PCEP session. This initialization consists in the exchange of an *Open* message, which carries various information for the establishment of the new connection, such as the *keepalive* timer, the *DeadTimer* and other policy or rules information that can specify conditions for the PCE path computation. If the establishment fails, because a peer does not respond before the establishment timer runs out or because both peers cannot agree on the session parameters, the TCP connection is closed. Several retries are possible, but the number of them are left to implementation.
If the establishment ends positively, the Keepalive messages are sent by both parties to maintain the communication up.
The first message that a peer has to receive right after the TCP establishment must be an *Open* message. Therefore, with a TCP connection already established, the peers both start a timer called *OpenWait* to wait for an *Open* message. If no *Open* message is received and the timer runs out, a PCEP error is triggered and the TCP connection is closed.
Any messages received that are not an Open must trigger the end of the TCP connection and a PCEP protocol error.
When an *Open* message has been sent by a peer, this peer starts a timer called *KeepWait*, at the end of which, if no *Keepalive* nor PCEP error message are received, the TCP connection is closed after a PCEP error is sent.

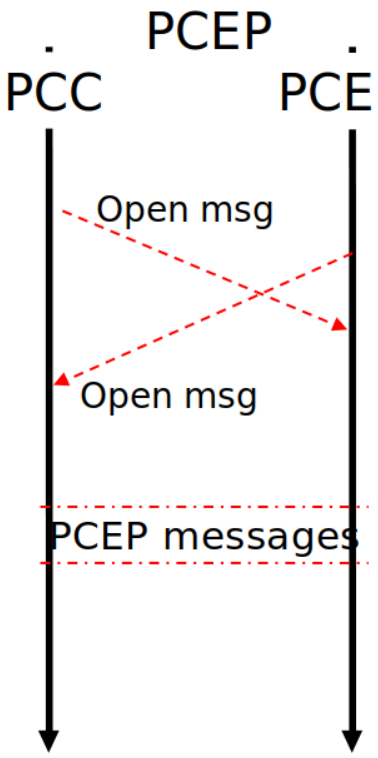

Figure 3.3: Initialization phase

3.3.2 Keepalive Session

After a PCEP session has been established, a peer has to know if the other party is still available for the communication. In order to do that, the PCEP protocol uses a *Keepalive* timer, a *DeadTimer* and the *Keepalive* message.
The *keepalive* timer is started by both peers and is restarted every time one party sends a message. After its expiration, the relative party sends a *Keepalive* message to the other.
When the PCEP session end, a *DeadTimer* is started by both peers; if no message is received after its expiration, the session is declared closed. This timer restarts every time a message is received in the ending session phase.

3.3.3 Path Computation Request

Once the PCEP session is established and well configured with at least one PCE, there can be an event that occurs and triggers the PCC to ask the PCE to compute a path or a set of paths with a path computation request.

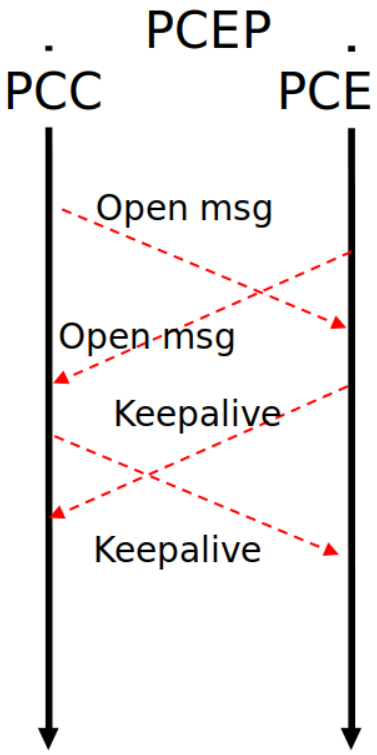

Figure 3.4: Initialization phase with keepalive

Therefore, the PCC chooses a PCE and sends a *PCReq* message that specifies the endpoints of the path and also the constraints and attributes for the path that the PCE has to find. The requests are identified by a request id number and addresses of PCE and PCC.
After the PCE have received the request, it starts the path computation, which can have two outputs:

- the computation has a solution: the PCE found at least path with all the constraints and requirements satisfied, hence it returns to the PCC the set of paths inside a PCEP reply (*PCRep*) message;

- the computation resolved in errors: the PCE could not find a path that satisfies the requirements and constraints given by the PCC, so the computation led to a failure; inside the *PCRep* message there is the reason of the failed path computation.

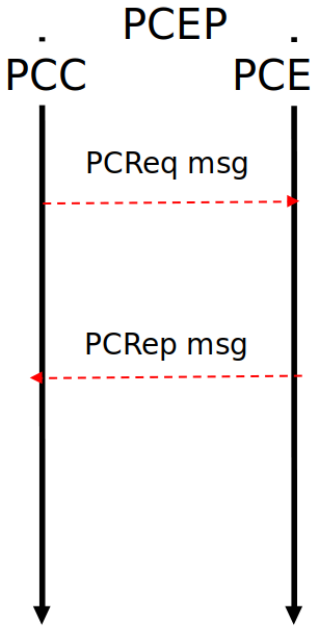

Figure 3.5: path computation request from a PCC to a PCE

3.3.4 Notification phase

This phase starts when a specific event occurs, hence the PCE or the PCC wants to notify of this event the other peer, since it may change the response for the other endpoint. For instance, a PCC that has already requested a path computation to a PCE but wants to cancel this request due to another event, can notify the same PCE with a Path Computation Notify (*PCNtf*) message to ask to erase the pending request. Another case could be a PCE, which has too much requests from different PCCs and wants to notify that a delay for the response can occur.
The figures 3.4 and 3.5 below show the packet flow of two possible cases.

3.3.5 Error handling

When some parameters of the request cannot be valid or a protocol error condition occurs, PCEP error messages are sent by peers to notify the type of error to the other party and which was the reason that produced the error: unknown type of message, malformed packet, parameter non supported, unknown policy, unexpected packet and many other reasons.
The figure 3.6 below shows the packet flow of the error handling.

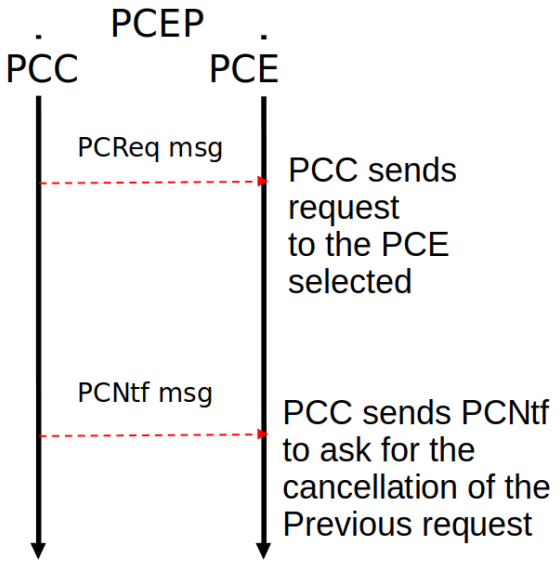

Figure 3.6: notification by a PCC to a PCE

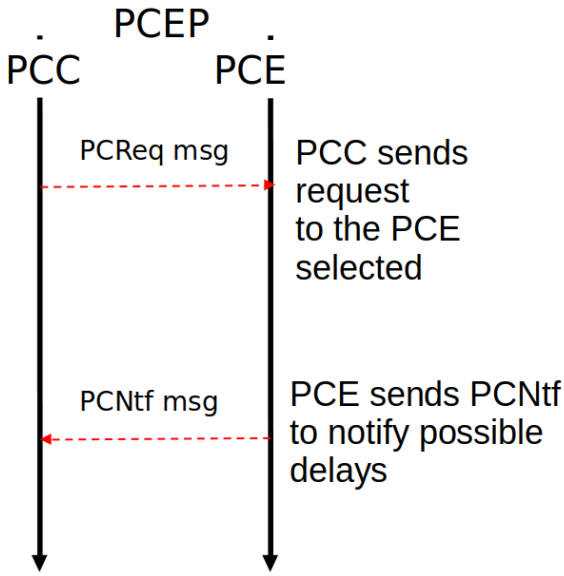

Figure 3.7: notification by a PCE to a PCC to notify delays

3.3.6 Ending session phase

A peer can end the communication and consequently the session whenever it wants by sending a PCEP *Close* message to the other peer, in order to notify the ending

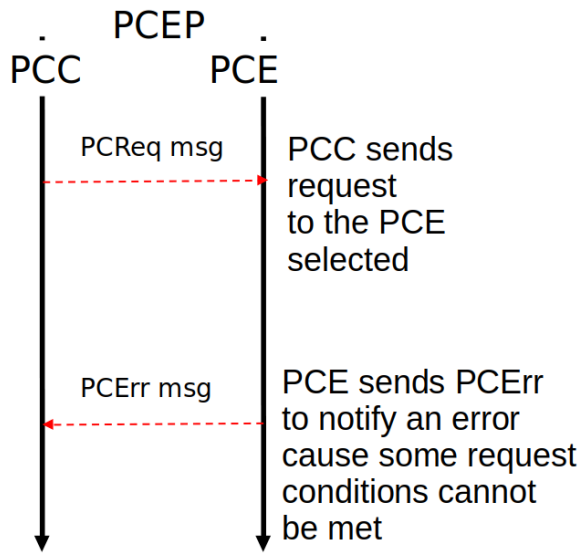

Figure 3.8: error handled by the PCE

of the session and then the closure of the TCP connection. When a PCE ends the session, the PCC clears all the pending requests sent before. when is the PCC to end the communication, the PCE does the same and clears the pending path computation requests received before.

In figure 3.7 there is an example of the packet flow.

3.4 Security Analysis

As we can see, security of the PCEP messages is not guaranteed by design. For instance, theoretically it is possible to change path computation responses in a malicious way that leads the PCC to set incorrect paths/LSPs. This paths can potentially redirect the network traffic to a part of the network under control of a malicious entity that has the means to destroy the service or manipulate the traffic.

3.4.1 Protocol vulnerabilities

To attack path computation responses there may be different possibilities: it is possible to intercept *PCRep* packets, manipulate them and resend them, since there is no authentication mechanism in PCEP protocol; similar to this approach, a malicious entity can intercept *PCReq* packets and manipulate them to force the PCE to make a different path computation from the original one; another way is to pretend to be a valid PCE and forge fake *PCRep* messages with incorrect LSPs

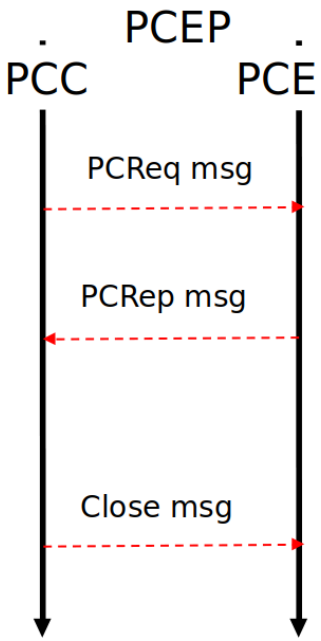

Figure 3.9: PCC ends the PCEP session after it received the reply from a previous request

inside.
It may be also possible to attack the PCE through different kinds of Denial of Service attacks (DoS); these attacks can make the PCE totally unavailable to resolve the path computation requested or just too much congested to answer back with acceptable delays for a peer. After all these consideration we can assume that confidentiality, integrity and authentication properties are not guaranteed by PCEP. So in general, PCEP protocol can be targeted by these kind of attacks:

- **Spoofing**: an entity impersonates a PCC or a PCE in order to manipulate or change the PCEP traffic;
- **Snooping**: an entity intercepts PCEP network traffic in order to manipulate it;
- **Falsification**: an entity forges fake PCEP messages;
- **Denial of service**: the normal functioning of a PCE is slow down or completed denied by an entity.

As we also said before and with the knowledge of the PCEP packets that we discussed in the previous section, we know that some of them (for instance, *PCReq*

and *PCRep* messages) contain vital information that show the topology of the network, the routing of the packets and as a consequence, where services related to specific packet flows are located. Therefore, it is really important to protect these information, since an attacker could easily understand by looking at them how services work, where they are located and where the traffic is routed and the prepare a variety of attacks to disrupt the network.

3.4.2 Proposed security techniques

Various techniques exist to improve security in PCEP protocol at different network levels. At TCP level, it is possible to use TCP-MD5 to protect the communication channel underlying PCEP, however we from [13] know that MD5 is prone to some limitations and insecurity that do not let to provide the right level of security that the network using PCEP protocol needs.
So the use of TCP-MD5 it should be only the basic security on which other more valuable and newer techniques lie on. Another possible technique for TCP is the TCP Authentication Option (TCP-AO), as stated in [14] whose implementation must support TCP-MD5 and take into account the transition time window where not all PCEP implementations deployed have the TCP-AO set, so policies for communication between peers are needed.
Also, a valuable technique that can be used as a solution could be the use of Transport Layer security (TLS) protocol. It gives protection against tampering, falsification, snooping and guarantees confidentiality . Opening a TLS channel for PCEP communication would secure the TCP channel underlying (only from the moment that the TLS channel starts) and all the PCEP important messages which may contain sensible information. The side effect of this solution is that would require to design and to implement the PCEP instantiation of a TLS channel, TLS handshaking and how to interpret TLS parameters.

Confidentiality

PCEP Privacy must be ensured, especially in a context like ours, where we are in a multi-domain environment and theoretically PCEP messages may pass through different autonomous systems and an attacker can intercept vital information from PCEP packets. In order to do that, we can use different approach:

- **IPsec**: it is possible to use IPsec tunnels to encrypt all TCP traffic an provide encryption. This may arise problems from an operational point of view and regarding configurations since theoretically PCEP is used in large scale networks. IPsec also gives authentication and integrity;
- **TLS**: TLS ensure encryption using pairs of public and private keys. Moreover it ensures the perfect forward secrecy if the option is enabled but, on the other

hand, all the peers using TLS need to have private and public key, something that is not so common.

Keys configuration

As we saw before, using TLS requires the configuration of keys for encryption ,but it is not the only technique that requires their use: in general for authentication, protection against tampering and encryption with other techniques, they are needed. If the network includes a small amount of PCCs and PCEs, than it is possible to manually configure key pairs and use them inside a session, but this is also prone to several vulnerabilities such as configuration errors and social engineering exploitation. However this approach is not feasible for large scale networks with many PCEs and PCCs, since it is too complex for operators to configure each PCE and PCC and keep every key pair updated. Also manually configured key pairs require manual updates and this is potentially risky from a security point of view. Another possible configuration for PCEs and PCCs is the use of group keys. An operator can generate and set for instance a group key for all the PCCs and PCEs belonging to the same Autonomous System (AS) or domain since they are in the same area of trust. This approach can significantly decrease the complex configuration work for operators; however, it is important to stress that the more are the entities that know the group keys, the more are the potential vulnerability risks which arise concerning their disclosure. Lastly with the group keys approach, the operators need to configure a different key for the communication between two PCEs, each from a different domain.
There are also further consideration to do for keys configuration regarding PCE discovery and its relation with the key exchange, but since it is out of the scope of this work, they are left to future works.

Denial of Service attacks

Denial of Service attacks are possible at TCP level but also in a already established PCEP session. Regarding TCP, the vulnerability are the same of many other protocol that run over TCP an a variety of attacks are possible (for instance the SYN attack). As mitigation for DoS attacks, PCEP implementations can provide features like the use of single registered port from which all the communications are expected and that does not allow multiple parallel connections from the same peer; another possibility is the creation of an access list that register all the authorized peers in order to avoid all the users that are not trusted.

3.4.3 Final considerations

After all the analysis, knowing the requirements for the experimental setup build, which will be shown in chapter 4 and 5, we decided to use the TLS approach to improve PCEP, since it covers almost all the security problems that the PCEP has and it is easier to implement from a technical point of view considering the type of PCE that we chose for the setup (java-based netphony-pce).

Chapter 4

Experimental Network Setup

In order to test and showcase the behaviour of the PCEP Communication Protocol and discuss about its security implication in a multi-domain environment, we need to build a setup with at least two separate domains, each of them with its own Path Communication Element (PCE). These PCEs must communicate in the beginning through the PCEP Communication Protocol for testing and analysing the communication. After that, a more secure communication protocol will be implemented, PCEPS.
Each domain will consists of routers that are able to support MPLS Segment Routing (SR-MPLS) and implement a Path Computation Client (PCC).
In the following sections will be given a more in depth descriptions on how the setup is build and the decisions that were taken.

4.1 Network Topology

The figure below (Figure 4.1) shows our experimental network setup composed of two domains. Each domain has four routers connected in a circle, with one of these connected to the domain's PCE.
The two PCEs are connected and they can communicate with each other. Lastly, there are two hosts connected respectively to router 1 (R1) and router 2 (R2) that can receive or generate network traffic.

4.1.1 Router Software Selection

The only requirements for the router software are the support for SR-MPLS or Segment Routing over IPv6 dataplane (SRv6) and the communication through the

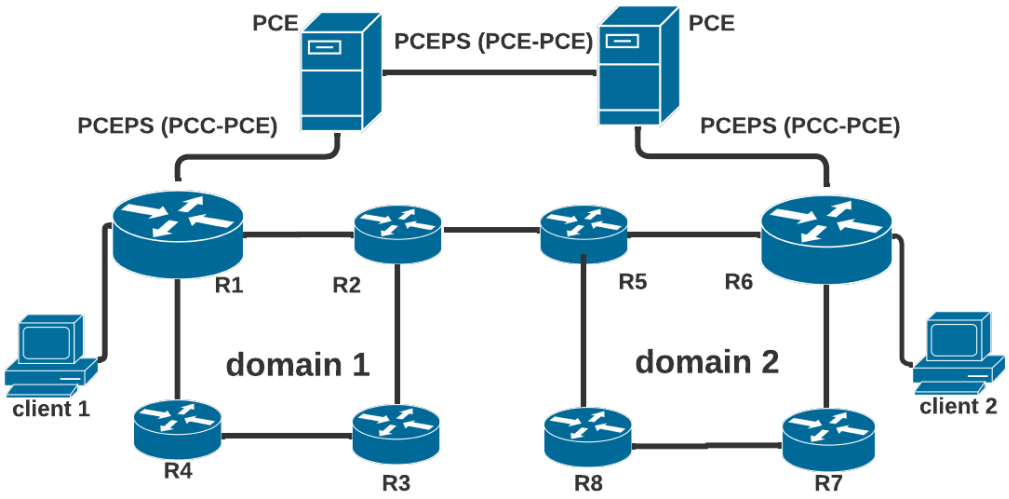

Figure 4.1: test setup with 2 domains and 2 PCEs.

PCEP Communication Protocol. Additionally we need a router that is able to behave as a PCC to receive routes from the PCE and delegate Label Switch Paths (LSPs) to the PCE.
The export of the topology information towards the PCE can be done using OSPF-TE, ISIS-TE or Border Gateway Protocol Link-State (BGP-LS). In the end we chose to adopt the Free Range Routing (FRR), a router open-source implementation that supports all the features needed and easily modifiable if needed.

4.1.2 Protocol Selection

The decisions that we made about protocol selections were taken based on past experimental setups, related to the UPIN framework, in order to use the results of this work also for other works that are out of the scope of this thesis. Therefore, we chose OSPF as Interior Gateway Protocol (IGP) and SR-MPLS for our underlay, since SRv6 implementation inside FRR was not finished.

4.2 Path Computation Element

For the Path Computation Element (PCE) deployment and implementation decisions, we used the same approach as for the protocol selections. They were based on past UPIN works to maintain the continuity with past works.

4.2.1 PCE Deployment

The only defined RFC for communication between PCEs is through the use of a Hierarchical Path Computation Element (H-PCE). This deployment type is not suitable for our requirements, since with this approach, the parent PCE needs a connection to each PCE where the path will traverse through. This method does not scale.
We need a distributed setup where each PCE of each domain only has to keep a connection to the PCE of the neighbouring PCEs. Segment Routing Traffic Engineering (SR-TE) policies are stateful, so the PCE needs to keep in memory a list of active policies.

4.2.2 TED population

The netphony-pce supports multiple ways of populating the TED, from BGP-LS, OSPF-TE or a predefined network description file. Since our domain topology is simple and the setup is based on another UPIN related experiment, we use the same method, that is using a description file to load the nodes, edges and SIDs into the PCE.

4.2.3 Path Allocation

The netphony-pce has a flexible algorithm plugin that allows to define custom path allocation algorithms.
Since currently there was not one implemented yet for segment routing with MPLS, we used a customized algorithm done for a past UPIN related setup.
It is a simplified algorithm based on [15]: when the PCE receives a computation request, it resolves the domain for both the destination and the start endpoints; when one of them is unknown, a no-path-possible error is sent back; if both endpoints exist in their domain, it computes a local path and sends it back; if the destination endpoint is not a member of the local domain but exists in the reachability manager and the latter has an entry to reach the PCE of this domain, it will forward the request to this PCE; if the domain has forwarded the request, it will wait for a response; if it gets a no-path-found response, this is sent back to the requester; if the PCE does not contain both the start and end destination node, but both have an entry in the reachability manager, it forwards the request to the PCE for the destination node and sends back the partial path to the requester.
This algorithm will take the shortest route possible between the two nodes. It disregards any other information or special requests.

4.2.4 PCE Chaining

RFC5441 [15] discusses the procedure on how to perform the Backwards Recursive Path Computation (BRPC) procedure to compute paths through multiple domains and the relevant extensions needed to the PCEP. The draft [16] extends stateful paths to interdomain deployments. Both RFCs leave the domain discovery and resolving to the implementation. Therefore, for this experimental setup we used the same simple approach of past works, that is to configure each PCE with an XML configuration file.

Chapter 5

Design considerations and technology evaluations

In this chapter we will introduce the evaluations that we did, following another assessment in a previous PCE related thesis work, in order to build a proper multi-domain Path Computation Element (PCE) setup. We will briefly explain the design choices that were made before the current work and now, and the technologies used inside the setup.
Later in this study, we will focus on the security issues of the PCEP protocol and what kind of approaches we wanted to use to improve the protocol on the working setup.

5.1 PCE technical requirements

Before presenting the experimental setup, how it is structured and how it works with the PCE, it is crucial to define what kind of technical requirements and features are needed for our PCE.
First we needed to perform path computation across multiple domains. So in order to do that, we needed a network that supports segment routing, therefore a PCE that implements at least [17] and [18] both for segment routing and stateful PCE.

5.1.1 Domain Discovery

Since currently there is not a proper defined way for a PCE to discover if another domain has a PCE or if there is a part of a domain controlled by a PCE, the only possible architectural solution is a PCEP extension, that is the Hierarchical Path Compute Element (H-PCE) extension. With this approach, the PCE in charge of a domain can share to a "parent" PCE, that is in charge of inter-domain PCE

communications, which OSPF, ISIS domain/Autonomous System (AS) it controls. But this method is unfeasible in a multi-domain setup, since it requires a central PCE that handles all domains and all of these must take a decision on who has the control over the central PCE. A possible solution to solve this issue is to build a setup where each domain run its parent PCE, with the latter that has connections to every domain-specific PCE. Lastly, we need support at a minimum level for H-PCE extension, just to signal another PCE to continue the path computation.

5.1.2 Function Chaining

It is mandatory for the architecture to be able to support encoding specific path requirements in the request, in order to request paths to pass through specific functions. A valuable option is the support for the Explicit Route Object (ERO), that allows to encode a sequence of AS numbers, areas or IP addresses where the path needs to go through at minimum. Thanks to this, we are able to explicitly request a set of functions if we already know their identifiers or their location.
Another option is the Objective Function Extension, which allows to ask the PCE to compute a path that matches the requirements of a specific Object Function.

5.1.3 Path Confidentiality

When a PCE receive a path allocation request, this PCE has to send back the complete segment list to spread it throughout the network. Nevertheless, it could happen that some operators do not want to lo leak the entire path. A way to overcome this issue is to encode the path in a generated segment list. Doing this, the path in the other domain will be encoded as a path key sub object.

5.2 PCE Evaluation

In this section we analyse the existing implementations of the PCE. For the evaluation, it was crucial to choose implementations that are open source or easily obtainable closed source, which implements the PCEP protocol, in order to adapt them to our needs and test them for different use case scenarios.
We also analysed which security features where already implemented, if there were any.

5.2.1 OpenDaylight Controller

OpenDaylight controller is an open source modular automation platform JVM, that supports a wide range of protocols to achieve vendor-agnostic network automation and control. Applications make use of northbound APIs exposed by the controller to

interact. Its goal is to be a one-stop-shop solution for Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) deployments.
Currently, in OpenDaylight the PCE support is in an active development status that for now include the initial Request For Comments (RFC) for the PCEP protocol; furthermore, it supports Segment Routing, stateful paths, binding labels, objective functions and route objects.
The OpenDaylight PCE has implemented some cryptographic features but does not have anything regarding a new PCEP secure protocol, neither regarding authentication and integrity of PCEP messages.

5.2.2 NorthStar Controller

NorthStar is the SDN controller released by Juniper Networks. It supports PCEP, but also NETCONF to interact with the routers in order to deploy paths. Furthermore it supports RSVP-TE. Regarding the PCE, it provides stateful paths, binding labels, segment routing extensions and support to initiate paths. Nothing concerning security has been implemented.

5.2.3 pceplib

Pceplib is an open source library that implements PCEP, developed by Volta Networks as a fork of libpcep, in turn developed by Acreo. The library is only a component to implement a fully functional PCE. Because of this it offers great flexibility for implementing extra standards, since it only handles the session management for PCEP. A big issue is that this library ships only one functional part implemented, that is the Path Computation Client (PCC), and implementing a fully functional PCE with a Traffic Engineering Database (TED) is out of the scope of the thesis, since we need an already functional one in order to study more in depth the PCEP protocol. Concerning security, nothing is implemented yet.

5.2.4 ONOS

ONOS is an open source network operating system that provides the control plane in an SDN. It provides the PCEP southbound interface to communicate with network devices. It supports the relevant standard for PCE Central Controller (PCE-CC) but on the other hand, no support for the H-PCE extension.

5.2.5 Cisco IOS XR

IOS XR is a networking operating system developed by Cisco. Based on the available documentation, it is not clear what features of SR-PCE are supported by

IOS XR; we can say it supports version 2 of the protocol and the relevant Segment Routing (SR) extensions. Cisco recommends a different kind of deployment for multi-domain setups.

5.2.6 Netphony-PCE

Used in multiple other papers to simulate PCE communication, also across multiple domains, netphony-PCE is a fully functional java based open source implementation of a PCE that follows the RFC 4655 architecture, and furthermore, it also implements a functional PCC.
It currently supports the following standards:

- Segment Routing;
- H-PCE extension;
- Path initiation.

There is a development branch, developed by Telefonìca, that supports some of the current RFCs, however almost nothing about the security is taken in consideration.

5.3 Comparison & Final Evaluations

Table 5.1: comparison between different PCEs

	OpenDaylight	NorthStar	ONOS	IOS XR	Netphony-pce
Open source	yes	no	yes	no	yes
H-PCE	no	no	yes	no	yes
SR support	yes	yes	yes	yes	yes
Stateful	yes	yes	yes	yes	yes
Extendable	yes	no	yes	no	yes
Full PCE	no	yes	yes	yes	yes
OSPF-TE	no	yes	yes	yes	yes
BGP-LS	yes	yes	yes	yes	yes
ISIS-TE	no	yes	yes	yes	no
Security	yes	no	no	no	no

5.3.1 PCE Implementation

Initially, the netphony-pce only implemented older drafts of the multiple PCE RFCs. However, the backend library that this PCE is using for the protocol implementation, called netphony-network-protocols, has a newer development version. This version implements the non-drafts Request For Comments (RFCs) that our setup needed for the communications.
In our porting to the non-draft RFCs version of the implementation, all the parts that the experimental setup didn't required were removed or changed: the multilayer and Generalized Multi-Protocol Label Switching (GMPLS) support were removed; specific objects, that were moved or renamed in later RFCs, were reshuffled.
However the netphony-network-protocols library has not implemented anything concerning the PCEP protocol security.

5.3.2 Comments

Based on the evaluation of implemented features that we need, as we can see from the comparison table above, Netphony-pce seems to implement most features and RFCs that are required and, since it is open source, we can eventually change the code to our needs and implement extra RFCs if we are missing specific required extensions. OpenDaylight could also be a valuable option, however the code is too much complex for our scope and not entirely open source. So we will explore more in depth the Netphony-pce in order to build a setup and analyse in this work the entire behaviour with a focus on the security issues and implication for the multi domain setup.

Chapter 6

PCEPS improvements and implementation

In this chapter we will show a possible implementation of PCEP with TLS, called PCEP secure (PCEPS), starting from an analysis of an old RFC draft never standardized [7] and then showing how it works inside the already presented experimental setup in chapter 4 together with considerations of improvings that it brings and possible new vulnerabilities that may arise.

6.1 Transport Layer Security (TLS)

TLS is currently the most widely adopted network security protocol. In the origin it was named SSL (Secure Socket Layer): this term should not be used anymore because the term refers to a protocol that has been discontinued because it was weak and had vulnerabilities. The version that has been later standardized is TLS. TLS creates a secure transport channel on top of layer 4 (also called session level security, even if it does not implement full session management) and it is also named layer 4.5 protocol. Its security features are:

- **Peer authentication**: it is compulsory for the server and optional for the client. If there is single peer authentication it is only the server, otherwise there is mutual peer authentication for both server and client. It is based on asymmetric challenge-response authentication so a proof of possession of private key is needed. For the server that is implicit because it is using its private key to create all the other keys for the other security features. On the contrary for the client there is an explicit signature performed and the user has no control over this. If peer authentication fails, the channel cannot be opened.

- **Message confidentiality**: it means that the content of each record transferred over TLS is optionally encrypted. It depends on the version of the protocol, in which in some cases is optional in others is mandatory.
- **Message authentication and integrity**: for each message sent over TLS channel these two properties are guaranteed. This type of authentication (data authentication) is different from peer authentication, because in this case is the proof that the data exchanged on the channel is really coming from the other peer that we are communicating with. By integrity we mean that the data in transit have not been manipulated, however this doesn't mean it prevents the manipulation.
- **Protection against replay and filtering attacks**: TLS provides replay protection, because if an old message is replayed another time, it is detected. In the same way it provides protection against message cancellation, which could change the meaning of the data that is exchanged. This kind of protection is obtained thanks to an implicit record numbering, which means that each message sent across TLS is a numbered record. This is because TLS is layered on top of TCP, which has the property of no data loss and it receives the record in the same order as they are sent. Since record number is used in message authentication, if there is a message received two times, then authentication fails. the same is for the cancellation of messages.

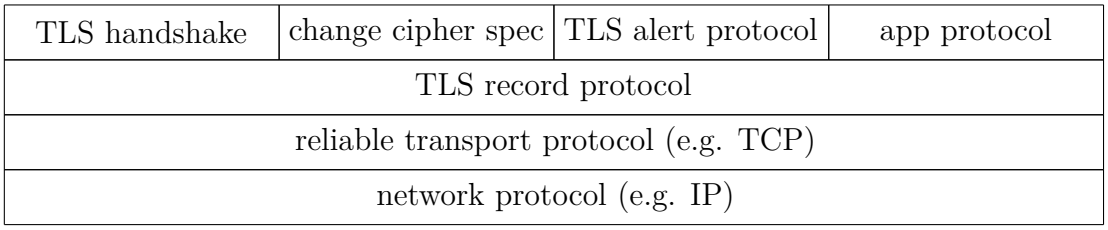

Figure 6.1: TLS protocol structure

The figure above shows how TLS protocol is structured: from an architectural point of view there is the basic network protocol, which in our case is IP protocol, then a reliable transport protocol is needed (again, in our case is TCP). On top of that there is the TLS record protocol, which encapsulates the data being transmitted. Of course then there is our application protocol which will be used inside TLS, that in our case will be PCEP protocol. Lastly there are TLS specific protocols, TLS handshake protocol, TLS change cipher spec protocol and TLS alert protocol. The handshake protocol is executed at the beginning when the TLS channel is created and it is the most vulnerable part, that is when an unprotected TCP channel is

transformed in a secured TCP channel; change cipher spec protocol is the one that can be used for changing keys or algorithms without closing the channel; the alert protocol is the one being used every time there is an issue.

6.2 Path Communication Element Protocol Secure (PCEPS)

As we saw in chapter 3, inside the communications with PCEP protocol there are interactions between the peers that can be critical for network operations and resources. So it is vital to keep secure all the PCEP architecture implementing security features accordingly to [19].
We analysed in chapter 3 the security issues and threats of the protocol and from RFC5440 we also have some proposal of security techniques that can be used against some major attacks. From RFC6952 instead we know that confidentiality is essential and must be ensured, since there can be the possibility that the two peers that are communicating through PCEP are in different Autonomous System (AS) and there is an higher risk for snooping attacks leaking sensitive information about the network.
As we said in the previous section, TLS provides message confidentiality, message integrity and peers authentication, which are the security feature most needed for PCEP. Moreover, we are using in the experimental setup the java-based netphony-pce and the library network-protocols made by TelefonicaId, so it is easier to implement TLS in the PCE and expand the already present PCEP protocol library to have the implementation of PCEPS.

6.3 PCEPS messages

In order to implement the Path Communication Element Protocol Secure, it is required to add only one new message for the protocol, that is the *StartTLS* message; all the other standard PCEP message are used once the TLS channel is established. Also new types of errors, encapsulated inside the PCEP *PCErr* message are defined. The *StartTLS* message packet is composed by the Common Header, with the Type field set to 13, followed by an empty body. The packet structure of the *StartTLS* message is as follows:

<StartTLS Message>::= <Common Header> .

6.3.1 New Errors

Table 6.1 shows the new types of errors introduced with PCEPS and their value with their related meanings.

Table 6.1: new error values introduced with PCEPS

Error-Type	Meaning	Value
25	PCEP StartTLS Failure	0: TBA
		1:received StartTLS after other msgs
		2:received other msg before StartTLS
		3:failure, no connection without TLS
		4:failure, connection without TLS ok
		5:no StartTLS before timer expired

6.4 PCEPS phases

In order to initiate a PCEPS session there are different phases to walk through:

- **Initialization and establishment** of a TCP connection;
- **StartTLS message** sent by both peers (PCE/PCC) to initiate TLS;
- **Negotiation** of TLS parameters and establishment;
- **PCEP messages over TLS** are sent.

Each phase follows the best practises for the TLS protocol showed in other RFCs for its standardization.

6.4.1 Initialization

Since peers using PCEP can communicate without TLS or using it and knowing that there is the transition problem (that will be analysed later), a new message is introduced to let a peer notify to the other that he wants to initiate a new secure TLS channel: the *StartTLS* message. That means that before the *open* message, but after the TCP establishment, the peers agree on a TLS session and then the normal PCEP messages are sent; in this way the TCP channel is secured even before starting the real PCEP session.
A peer that wants to communicate with PCEPS instead of PCEP must start first the *StartTLSWait* Timer to keep waiting from the other peer a *StartTLS* message and not the *OpenWait* Timer.

How a PCEPS peer may discover if there are others that can communicate with PCEPS is out of the scope of this work and is left for future work.
It is important to stress that it is not possible to secure an already opened PCEP session, but it is necessary to close the ongoing session and re-create another one using the proper methods and messages described later.
As we said, for the initialization phase of PCEPS, after the TCP establishment, a peer (so a PCC or a PCE) first has to send to the other party a *StartTLS* message to notify its intention of opening a TLS channel; if the other peer agrees, it will answer back with another *StartTLS* message. If a peer is configured to operate only with TLS, it can send the *StartTLS* immediately after the TCP establishment, otherwise it has to wait if the other peer's intentions, thus waiting for an *Open* or a *StartTLS* message. When the peer receives a *StartTLS* message and has sent its own, the TLS establishment can start. In figure 6.2 we can see the packet flow for the initialization phase.
So in the end, the first message right after the TCP establishment is the *StartTLS*,

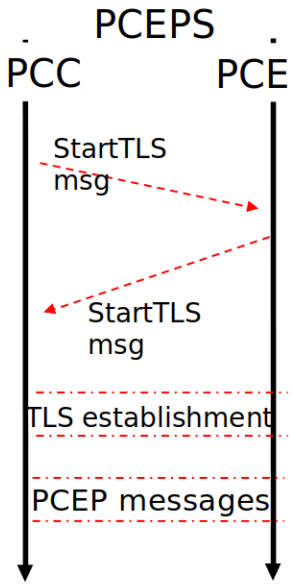

Figure 6.2: PCEPS initialization phase

and if any other PCEP message is received before the *StartTLS* must be handled like an unexpected message and therefore a proper *PCErr* message will be sent or received with the consequence of the closure of the TCP connection.

6.4.2 Possible Use Cases

The configuration of the PCE/PCC that speaks PCEPS regarding how to handle a new PCEPS session, leads to different use cases based on if the strict TLS option is enabled or not and if a peer can speak PCEPS or not.
Additionally new error cases are introduced to handle and notify these kind of issues.

Use Case: Strict TLS

If both peers (for instance, a PCC that wants to communicate with a PCE) are configured to only speak PCEPS, each peer sends the *StartTLS* right upon the establishment of the TCP connection without waiting first the other peer.
Figure 6.3 shows the packet flow for the initialization of a PCEPS session between a PCC and a PCE with both the strict TLS enabled. When each of them has both sent and received *StartTLS* messages then the normal TLS establishment can start and when this ends successfully, they can begin the secured PCEP communication.

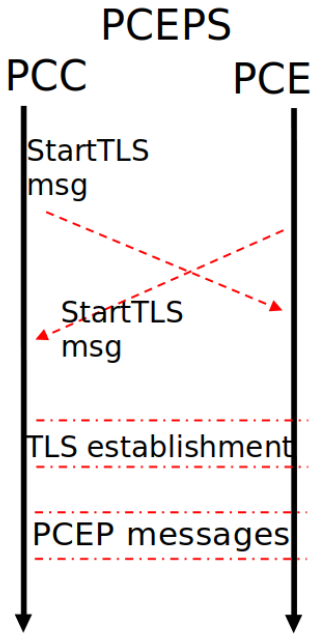

Figure 6.3: both PCE and PCC communicate only with TLS so right after the TCP establishment they both send a *StartTLS* message

Use Case: one peer speaks PCEPS, the other one does not

In this case we have a PCC, which can speak both PCEP with TLS and standard PCEP, that tries first to open a PCEPS session with the PCE, which on the contrary cannot communicate with PCEP over TLS. So first the PCC sends a *StartTLS* message to the PCE to notify its intention. The PCE on the other hand cannot process the *StartTLS* message, so it handles it like an unknown packet: it sends a *PCErr* message to notify its error to the PCC. As a consequence the PCC immediately closes the TCP connection after having sent a *Close* message.
At this point the PCC, since is configured to communicate also with PCEP if PCEPS is unabled, re-establishes the TCP connection with the PCE and sends a normal *Open* message to open a standard PCEP session with the other peer. The PCE finally sends its own *Open* message to the PCC and a standard PCEP session begins.
The figure 6.4 shows the packet flow of this case.

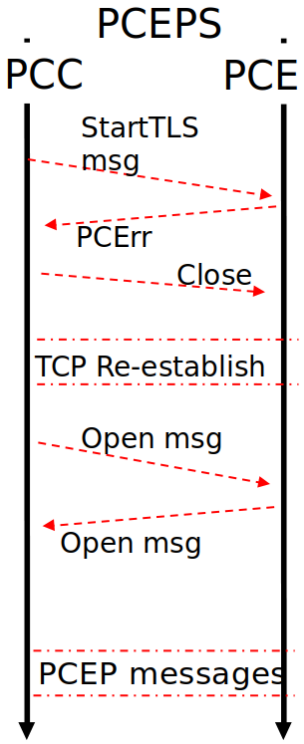

Figure 6.4: PCC which speaks both PCEP and PCEPS sends *StartTLS* msg to open a PCEPS session but PCE does not speaks PCEPS

Use Case: both peers speak PCEPS but cannot establish a TLS channel

Here we find a PCC that sends to a PCE the *StartTLS* to notify its intention to open a secure channel and start to communicate through PCEPS; the PCE responds with its own *StartTLS* message, then they agree on the TLS session parameters during the TLS establishment. In the meantime, an error occurs during this establishment, so they immediately close the TCP connection. Eventually, if both peers are configured to enable the retries of the establishment, they will re-open a TCP connection and retry to establish a TLS channel for a number of time equivalent to the value configured inside each peer. Otherwise, if they are unable to perform retries, they will no longer communicate with each other and the other party will be signed as no longer available. The latter situation occures only if the peers are strict TLS, otherwise they will maybe try to open a standard PCEP session.
In figure 6.5 it is shown the packet flow of this case.

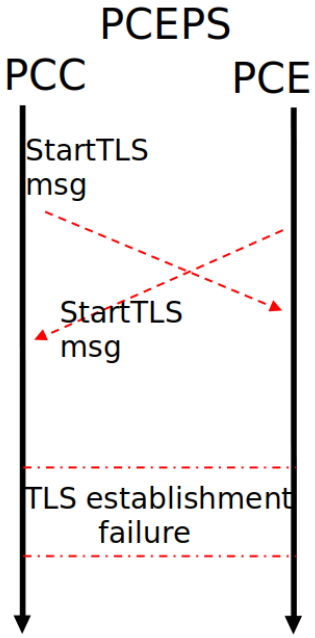

Figure 6.5: PCC and PCE agree to open a TLS channel but they cannot due to a failure

Use Case: a peer speaks only PCEPS, the other one only PCEP

The PCC that is pre-configured to operate with strict TLS only sends the *StartTLS* message to open a secure channel with the PCE and communicate through PCEPS.

The PCE is not able to understand PCEPS, so it treats the *StartTLS* as an unknown packet. Then it sends a *PCErr* message to notify the internal error and after that, it sends a *Close* message to the PCC to close the session and the TCP connection. Since The PCC operates with strict TLS, it will not try to re-open a session with the PCE through standard PCEP.
The packet flow is shown in figure 6.6.

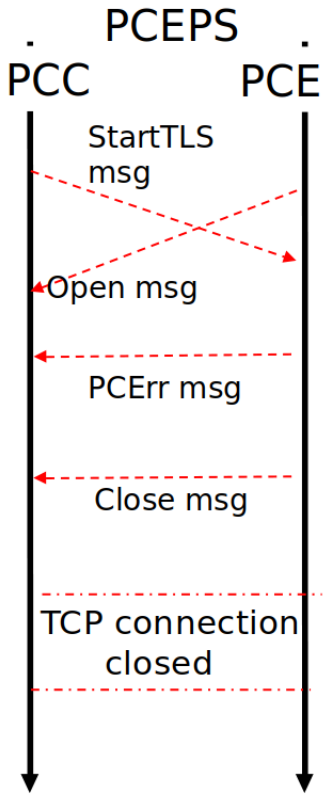

Figure 6.6: PCC sends a *StartTLS* to open a secure PCEPS session but PCE speaks PCEP only, so it closes the connection without retries with PCEP

6.4.3 Negotiation of TLS parameters and establishment

After the Initialization phase of a PCEPS session between the two peers is finished, the TLS establishment can start, accordingly to TLS connection establishment procedure.
There are some feature that the peers has to support: first the minimum TLS version supported must be TLS 1.2 or above; certificate based authentication is required for peer authentication; a proper ciphersuite for integrity and confidentiality is required

and moreover, TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 has to be supported at minimum.
Peer authentication is done following two possible models:

- X.509 with Public Key Infrastructure (PKI): a list of trusted Certification Authorities (CAs) must be included inside the implementation and the validation of certificates must follow all the standard and best practices for PKI; revocation methods inclusion inside PCEPS is optional but recommended as best practise;
- X.509 with certificate fingerprints: there can be a list of trusted certificates in the implementation, in order to identify peers; these certificates are identified by a fingerprint encoded using Distinguished Encoding Rules (DER).

Finally after the TLS connection parameter have been decided and the TLS channel is established, the two peers can start to exchange PCEP packets; first the peers start the *Openwait* Timer and send their own *Open* message. If everything goes as expected, a PCEPS connection is established correctly.

6.4.4 TLS establishment Failure

If the negotiation phase of TLS parameters fails or the peer authentication resolve in errors for whatever reason, the peers need to close the connection immediately. If a peer is configured for connection retries, then it can start the retry procedure.

6.5 The Transition Problem

Everything new that is implemented with PCEPS works just fine without interfere with the normal PCEP mechanisms. That means that a PCEPS user can establish a work flow with a standard PCEP user without disrupt the connection or trigger unexpected behaviours, as we saw in the previous sections.
This means that everything will still work during the so called transition time period, a time window where upgraded PCEPS devices have to coexist with standard old PCEP devices that gradually will be upgraded with new versions supporting PCEPS. So a PCEP device should always accept a connection and sessions with or without TLS until every peer in the network is able to speak PCEPS. For instance, a PCE will wait for a connection request from a PCC and will answer back based on the first message received: if it is an *Open* message, it will allow a standard PCEP session, otherwise if it is a *StartTLS* message, it will set up a secure channel to communicate through PCEPS.
If a PCC supports the communication both with PCEPS and PCEP , it will try first to open a secure channel sending a *StartTLS* message to the other peer;

if the latter is unable to speak PCEPS, it will close the connection and may try to establish a standard PCEP session.

6.5.1 Downgrade attacks

Now we consider the case related to the transition time period where a PCC and a PCE that can communicate with or without TLS, want to open a PCEPS session. Normally, the PCC sends to the PCE a *StartTLS* message to notify its intention and the latter will answer back with another *StartTLS* message to start the TLS establishment. But we know that if a peer wants to communicate with another one that is unable to open a PCEPS session, the communication after the *StartTLS* from the first peer will resolve in a failure with the second peer that sends back a *PCErr* message with error type value 1 (received malformed *Open* message or a non *Open* msg). After that, the PCC may retry to establish a standard PCEP session: this introduce a possible vulnerability to the system that can be exploited.

Use case 1: an attacker performs a MITM attack and inject a *PCErr* msg

There is a possibility that an attacker performs a MITM (Man-in-the-middle) attack and so gains undetected access to the packets in transit between for instance a PCC and a PCE that are communicating with each other.
In this particular case the man-in-the-middle can intercept the packets, manipulate the and even inject fake PCE/PCC responses in order to control the communication flow without being noticed. In figure 6.7 is depicted a possible attack where the attacker is placed undetected between a PCC and a PCE trying to open a secure channel and establish a PCEPS session. However the attacker wants to sniff the packets and steal sensitive data encapsulated inside PCEP packets (e.g. computed paths to see where some particular services are located inside the network), which in a secure channel would be inaccessible due to the encryption. So as depicted in the figure below, he intercepts the two *StartTLS* messages from both PCC and PCE and inject inside the communication fake *PCErr* messages to force the use of standard PCEP, pretending to be the endpoint of the communication from the PCE/PCC point of view. The PCC and PCE receive the fake *PCErr* messages and then they start the closing procedure. Since they are configured to retry the PCEP session establishment without TLS in case of impossibility of a secure channel, they will open a Standard PCEP session. Now the attacker is able to sniff the packets and steal all the data in clear without being noticed.
That is why a network using PCEPS should use the strict TLS option enabled.

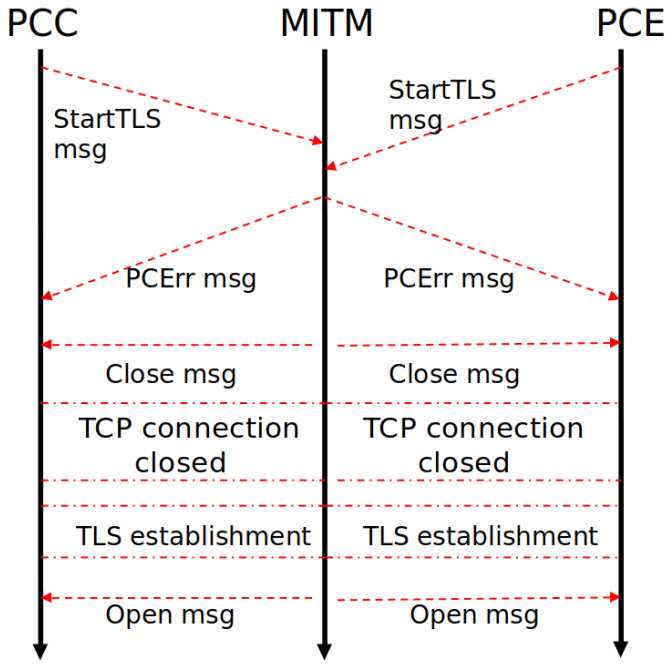

Figure 6.7: a MITM inject a *PCErr* msg in order to downgrade from PCEPS to PCEP

Use case 2: TLS downgrade attack

Using [20] and [21], we know that there are various versions of TLS: the original SSL, version 2, 3 and then TLS 1.0, 1.2 and 1.3. Typically the older versions have security problems and should not be used unless explicitly required.
Normally, client sends within *ClientHello* message the highest supported version and the server notifies (in *ServerHello*) the version to be used, which is the highest in common with the client.
A standard version negotiation between client and Server is like the following:

- **Agreement on TLS 1.2**
  - (C →S) 3,3 (e.g. major version 3, minor version 3, which is TLS 1.2 by default)
  - (S →C) 3,3 (server agrees)
- **Fallback to TLS 1.1**
  - (C →S) 3,3
  - (S →C) 3,2 (Server downgrades to a previous version)

Here comes the problem: some servers, rather than sending the correct response, close the connection; then the client has no choice but to try again with a lower protocol version.
This raises the Downgrade attack: the attacker perform a MITM (Man-in-the-middle attack) and then sends fake server response, to force repeated downgrade until reaching a vulnerable version (e.g. SSL-3) and then execute a suitable attack. So the implementation of PCEP over TLS should follow the best practises listed in [22].
In figure 6.8 is shown a schema of how the TLS downgrade attack works.

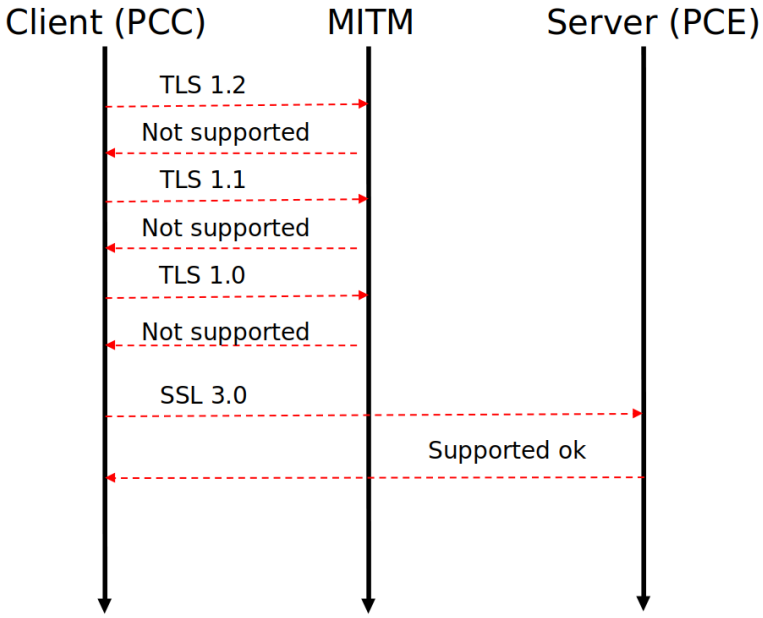

Figure 6.8: MITM performs a TLS downgrade attack during TLS negotiation

6.6 Experimental implementation

For the implementation of all the PCEPS features mentioned in the previous sections, as we said we started first from the Netphony-network-protocols library, developed by TelefonicaId, that is a java-based open source collection of libraries containing the code implementation for many network protocols, in particular PCEP but also other protocols, for instance OSPF, which we needed for our experimental setup.
Afterwards, we worked on the PCE/PCC code, that is the Netphony-pce (evaluated in chapter 3 for the experimental setup), a java-based implementation of a PCE (that also works as a PCC): here we added all the features needed to support

PCEPS.
All the source code, documentation and scripting used for simulations and test for our work can be found in a public repository[1].

6.6.1 Netphony-network-protocols library

The netphony-network-protocols implements four networking protocols: PCEP protocol, BGP-LS protocol and support for BGP-LS-TE, RSVP-TE and OSPF-TE with OSPF version 2.
We mainly worked on the PCEP protocol adding new features to support PCEPS. In particular, different changes were made in the source code:

- a file named PCEPStarttls.java located in
  netphony-network-protocols/src/main/
  /src/main/java/es/tid/pce/pcep/messages/
  is added: it contains a new java class that build the new message type StartTLS to the implementation;

```
1     public class PCEPStarttls extends PCEPMessage {
2     //No elements inside
3
4     /**
5      * Create new Starttls
6      */
7     public PCEPStarttls() {
8         this.setMessageType(PCEPMessageTypes.MESSAGE_STARTTLS);
9     }
10
11    public PCEPStarttls(byte[] bytes) throws PCEPProtocolViolationException {
12        super(bytes);
13    }
14
15    public void encode() throws PCEPProtocolViolationException {
16        this.setMessageLength(4);
17        this.messageBytes=new byte[this.getLength()];
18        encodeHeader();
19    }
20 }
21
```

- in the same location of PCEPSStarttls, inside the file PCEPMessageTypes.java, which contains constant variables to define the types of different PCEP messages, we added the definition of a new constant, that is the new message type *StartTLS* with the value 13 as defined in the previous sections:

[1]https://bitbucket.org/leoboldrini/workspace/repositories

```
1    public static final int MESSAGE_STARTTLS = 13;
2
```

- minor changes were also made to add the new error values of PCEPS.

6.6.2 Netphony-pce

For the netphony-pce we did major changes on how the PCE session works and on the behaviour of the PCE/PCC during the initialization phase. The TLS version used for this implementation is the 1.3.
Here we list all the major changes:

- inside the file PCEPValues.java we added for testing purpose the constant variable that is used inside the code of the netphony to trigger the new implementation of PCEPS as we will see later and another constant that define a new state, the *StartTLS wait* state, in which the pce is waiting for a *StartTLS* message from the other peer;

```
1        public static final int PCEP_STATE_STARTTLS_WAIT = 5;
2        public static final int PCEP_TLS=1;
3
```

- we created a file, StartTlsWaitTimer.java, to define a the new timer for the *StartTLS wait* state following the guidelines discussed in the previous sections;

- we decided to easily test our setup to manually generate a key pair and a certificate for both PCC and PCE to be used during the TLS establishment phase for the authentication;

- inside GenericPCEPSession.java, we defined a new method that handles the creation and setting up of the TLS context with the keys and certificate configurations: we create an SSLSocket and configured with TLSv1.3 and for encryption, we enabled the algorithm AES_128_GCM_SHA256;

- again inside the file GenericPCEPSession.java, which contains the code that handle and menage new PCEP sessions both client side (PCC) and server side (PCE), we rewrite the portion of the code that handles the initialization of a new PCEP session, changing the behaviour accordingly to the guidelines discussed in the previous sections; in this portion of the code the *StartTlsWait*, *OpenWait* and *KeepWait* timers are instantiated, then if the *starttlsOn* value

```
1   public class StartTlsWaitTimer extends TimerTask {
2
3   //  private DataOutputStream out=null;
4   //Use this to send messages to peer
5
6   private PCEPSession parentPCESession;
7   private Logger log;
8
9   public StartTlsWaitTimer(PCEPSession parentPCESession) {
10      this.parentPCESession = parentPCESession;
11      log = LoggerFactory.getLogger("PCEServer");
12  }
13
14
15  public void run() {
16      log.warn("STARTTLS WAIT Timer OVER");
17      PCEPError perror = new PCEPError();
18      PCEPErrorObject perrorObject = new PCEPErrorObject();
19      perrorObject.setErrorType(ObjectParameters
20      .ERROR_ESTABLISHMENT);
21      perrorObject.setErrorValue(ObjectParameters
22      .ERROR_ESTABLISHMENT_NO_STARTTLS_MESSAGE);
23      ErrorConstruct error_c = new ErrorConstruct();
24      error_c.getErrorObjList().add(perrorObject);
25      perror.setError(error_c);
26      log.info("Sending Error");
27      parentPCESession.sendPCEPMessage(perror);
28      this.parentPCESession.killSession();
29      return;
30     }
31  }
32
```

Figure 6.9: StartTlsWaitTimer

```
public SSLSocket CreateTLSContext() throws Exception
   {
       try {
           SSLContext context = SSLContext.getInstance("TLS");
           KeyManagerFactory keyManagerFactory = //
           KeyManagerFactory.getInstance("SunX509");
           KeyStore keyStore = KeyStore.getInstance("JKS");

           keyStore.load(new FileInputStream(
           "/vagrant/temp/keystore.test"), //
           "storepass".toCharArray());
           keyManagerFactory.init(keyStore, "storepass"
           .toCharArray());
            TrustManagerFactory trustManagerFactory = //
            TrustManagerFactory.getInstance("SunX509");
           KeyStore trustStore = KeyStore.getInstance("JKS");
           trustStore.load(new FileInputStream(
           "/vagrant/temp/truststore.test"), "storepass"
           .toCharArray());
           trustManagerFactory.init(trustStore);
           context.init(keyManagerFactory.getKeyManagers(), //
           null, null);
           SSLSocketFactory sslSf = context.getSocketFactory();
           SSLSocket secureSocket= //
           (SSLSocket) sslSf.createSocket(this.socket,
           this.socket.getInetAddress().getHostAddress(),
           this.socket.getPort(), true);
           secureSocket.setEnabledCipherSuites(
           new String[]{ "TLS_AES_128_GCM_SHA256" });
           return secureSocket;
           }catch(GeneralSecurityException e)
           {
               throw new IOException(e);
           }
   }

```

Figure 6.10: createTLSContext

```
1     protected void initializePCEPSession(
2          boolean zeroDeadTimerAccepted,
3          int minimumKeepAliveTimerAccepted,
4          int maxDeadTimerAccepted,
5          boolean isParentPCE, boolean requestsParentPCE,
6          Inet4Address domainId, Inet4Address pceId,
7          int databaseVersion, boolean isClient
8          )
9     {
10        remotePeerIP = (Inet4Address) socket.getInetAddress();
11        /**
12         * Byte array to store the last PCEP message read.
13         */
14        byte[] msg = null;
15        //First get the input and output stream
16        try {
17            out = new DataOutputStream(socket.getOutputStream());
18            in = new DataInputStream(socket.getInputStream());
19        } catch (IOException e) {
20            log.info("Problem in the sockets,
21            ending PCEPSession");
22            killSession();
23            return;
24        }
25        pcepSessionManager.notifyPeer((Inet4Address) socket.getInetAddress
      ());
26
27        StartTlsWaitTimer swt = new StartTlsWaitTimer(this);
28        OpenWaitTimerTask owtt = new OpenWaitTimerTask(this);
29        KeepWaitTimerTask kwtt = new KeepWaitTimerTask(this);
30
31        //value just to try pce with tls set on.
32        //set this value to 0  PCEPValues.PCEP_TLS to change to normal pcep
33        int starttlsOn=PCEPValues.PCEP_TLS;
34        if(starttlsOn>0)
35        {
36            this.setFSMstate(PCEPValues.PCEP_STATE_STARTTLS_WAIT);
37            log.info("Entering PCEP_STATE_STARTTLS_WAIT");
38            log.info("Scheduling STARTTLS Wait Timer");
39            this.timer.schedule(swt, 60000);
40            log.info("Sending STARTTLS msg");
41            PCEPStarttls p_starttls_snd=new PCEPStarttls();
42            sendPCEPMessage(p_starttls_snd);
43        }
44        else {
45            //set up and send the first open message of the session;
46            initializeOpen(swt, owtt, kwtt, isParentPCE,
47            requestsParentPCE, domainId, pceId,
48            databaseVersion);
49        }
50        ...
51    }
52
```

Figure 6.11: initializePCEPSession

(took from the constant value in the PCEPValues.java file) is set >0, then the FSM state is set to the value PCEP_STATE_STARTTLS_WAIT (again defined in PCEPValues.java); then the *StartTlsWait* timer is scheduled for 60000 ms and a new *StartTLS* message is created and send to the other peer;

- in this portion of the code there is the handling of the received packets; inside the try block we have different lines of code that are commented. These are the version of the code that can be use when uncommented for encryption when the PCE is inside a setup where is impossible to establish a TLS connection, but encryption is mandatory. Note that AES GCM not only encrypts, but also guarantees integrity of the packets:

```
 //Now, read messages until we are in SESSION UP
while (this.FSMstate != PCEPValues.PCEP_STATE_SESSION_UP)
{
try {
if(tls_enabled==true){
  //msg=readEncMsg(in, cipherDec);

 // byte[] iv = new byte[16];
 // random.nextBytes(iv);
 // gcmSpec = new GCMParameterSpec(128, iv);
 // cipherEnc = Cipher.getInstance("AES/GCM/PKCS5Padding");
 // cipherEnc.init(Cipher.ENCRYPT_MODE, new SecretKeySpec("0123456789abcdef".
    getBytes(StandardCharsets.UTF_8), "AES"), gcmSpec);
 // cipherDec = Cipher.getInstance("AES/GCM/PKCS5Padding");
 // cipherDec.init(Cipher.DECRYPT_MODE, new SecretKeySpec("0123456789abcdef".
    getBytes(StandardCharsets.UTF_8), "AES"), gcmSpec);
 // this.sslIn=new CipherInputStream(sslInputStream, cipherDec);
 // this.sslOut=new CipherOutputStream(sslOutputStream, cipherEnc);
 // msg=readMsgEnc(sslIn);
  msg = readMsg(in);
}
else{
  msg = readMsg(in);
}
} catch (IOException e) {
    log.info("Error reading message, ending session" +
    e.getMessage());
    killSession();
    return;
    }
    if (msg != null)
    {//If null, it is not a valid PCEP message
    switch (PCEPMessage.getMessageType(msg)) {
    case PCEPMessageTypes.MESSAGE_STARTTLS:
    log.info("STARTTLS message received");
    if(this.FSMstate==PCEPValues \\
      .PCEP_STATE_STARTTLS_WAIT)
    {
        PCEPStarttls p_starttls;

        try {
        p_starttls=new PCEPStarttls(msg);
        log.debug(p_starttls.toString());
```

```
42          swt.cancel();
43
44          try {
45          sslSocket= CreateTLSContext();
46          if(isClient==false)
47              sslSocket.setUseClientMode(false);
48          this.socket=sslSocket;
49          this.sslInputStream =sslSocket.getInputStream();
50          this.sslOutputStream =sslSocket.getOutputStream();
51
52          //parameters for aes gcm encryption
53          //byte[] iv = new byte[16];
54         // random.nextBytes(iv);
55          //gcmSpec = new GCMParameterSpec(128, iv);
56          //cipherEnc = Cipher.getInstance("AES/GCM/PKCS5Padding");
57          //cipherEnc.init(Cipher.ENCRYPT_MODE, new SecretKeySpec("0123456789
    abcdef".getBytes(StandardCharsets.UTF_8), "AES"), gcmSpec);
58          //cipherDec = Cipher.getInstance("AES/GCM/PKCS5Padding");
59          //cipherDec.init(Cipher.DECRYPT_MODE, new SecretKeySpec("0123456789
    abcdef".getBytes(StandardCharsets.UTF_8), "AES"), gcmSpec);
60          //this.sslIn=new CipherInputStream(sslInputStream, cipherDec);
61          //this.sslOut=new CipherOutputStream(sslOutputStream, cipherEnc);
62          //this.out = new DataOutputStream(sslOut);
63          //this.in = new DataInputStream(sslIn);
64          this.out = new DataOutputStream(((SSLSocket)socket).getOutputStream())
    ;
65          this.in = new DataInputStream(((SSLSocket)socket).getInputStream());
66          tls_enabled=true;
67          log.info("SECURE SOCKET OPENED WITH TLS 1.3");
68          initializeOpen(swt, owtt, kwtt, isParentPCE, requestsParentPCE,
    domainId, pceId, databaseVersion);
69          }catch(Exception e) {
70              e.printStackTrace();
71          }
72      }catch (PCEPProtocolViolationException e1) {
73          log.info("Malformed STARTTLS, INFORM ERROR and close");
74          e1.printStackTrace();
75          PCEPError perror = new PCEPError();
76          PCEPErrorObject perrorObject = new PCEPErrorObject();
77          //FIX: change error object parameter adding starttls error type
78          perrorObject.setErrorType(ObjectParameters.ERROR_ESTABLISHMENT);
79          perrorObject.setErrorValue(ObjectParameters.
    ERROR_ESTABLISHMENT_INVALID_OPEN_MESSAGE);
80          ErrorConstruct error_c = new ErrorConstruct();
81          error_c.getErrorObjList().add(perrorObject);
82          perror.setError(error_c);
83          log.info("Sending Error and ending PCEPSession");
84          sendPCEPMessage(perror);
85          pcepSessionManager.notifyPeerSessionFail((Inet4Address) this.socket.
    getInetAddress());
86          killSession();
87      }
88
89
90      }
91      else
92      {
93          log.info("ignore STARTTLS message, already one received");
94      }
95      break;
96
```

If the state is different from PCEP_STATE_SESSION_UP and is equal to PCEP_STATE_STARTTLS_WAIT, the PCE/PCC reads the received message and if it is of type MESSAGE_STARTTLS then it will stop the *StartTlsWait* timer, try to open a secure socket and start the *Open* message initialize procedure. If a *StartTLS* message was already received, then this new *StartTLS* will be just ignored;

- we also added code for handling and ignore possible *StartTLS* messages received when the session is already up in many PCEP session files (for Domain PCE session, PCC session and server parent PCE session):

```
while (this.FSMstate == PCEPValues.PCEP_STATE_SESSION_UP) {
try {
if(tls_enabled==true){
  //msg=readEncMsg(in, cipherDec);
 // byte[] iv = new byte[16];
 // random.nextBytes(iv);
 // gcmSpec = new GCMParameterSpec(128, iv);
 // cipherEnc = Cipher.getInstance("AES/GCM/PKCS5Padding");
 // cipherEnc.init(Cipher.ENCRYPT_MODE, new SecretKeySpec("0123456789abcdef".
    getBytes(StandardCharsets.UTF_8), "AES"), gcmSpec);
 // cipherDec = Cipher.getInstance("AES/GCM/PKCS5Padding");
 // cipherDec.init(Cipher.DECRYPT_MODE, new SecretKeySpec("0123456789abcdef".
    getBytes(StandardCharsets.UTF_8), "AES"), gcmSpec);
 // this.sslIn=new CipherInputStream(sslInputStream, cipherDec);
 // this.sslOut=new CipherOutputStream(sslOutputStream, cipherEnc);
 // msg=readMsgEnc(sslIn);
  msg = readMsg(in);
}
else{
  msg = readMsg(in);
}catch (IOException e) {
//cancelDeadTimer();
//cancelKeepAlive();
//timer.cancel();
//try {
//in.close();
//out.close();
//} catch (IOException e1) {
//}
    log.warn("Finishing PCEP Session abruptly");
    this.killSession();
    return;
    }
    if (this.msg != null) {
    //If null, it is not a valid PCEP message
        boolean pceMsg = true;
    //By now, we assume a valid PCEP message has arrived
//Depending on the type a different action is performed
        boolean STARTTLSmsg=false;
        switch (PCEPMessage.getMessageType(this.msg)) {

            case PCEPMessageTypes.MESSAGE_OPEN:
                log.debug("OPEN message received");
           //After the session has been started,ignore subsequent OPEN
    messages
```

```
43                  log.warn("OPEN message ignored");
44                  break;
45              case PCEPMessageTypes.MESSAGE_STARTTLS:
46                  log.debug("STARTTLS message received");
47            //After the session has been started,ignore subsequent STARTTLS
    messages
48                  log.warn("second STARTTLS message
49                  received");
50
51                  break;
52              ...
53          }
54      ...
55      }
56  ...
57  }
58
```

- another work that we did was also on the readMsg() function: we optimized the normal function that was already present in the code and, based on this one, we added two new functions called readEncMsg(), which reads an encrypted message from the SSLSocket and makes the deciphering procedure to give the deciphered message to the instructions that read the header and all the message fields to reconstruct the message object, and readMsgEnc() which takes a CipherInputStream as a parameter and uses that instead of the DataInputStream to perform the deciphering; this two functions where used as test to see if the chosen encryption algorithm could be used for the PCEP message format. However they can also be used if, as mentioned before, the TLS connection is impossible to use in a setup but still the encryption is mandatory. In our case using TLS, the normal readMSg() function can be used:

```
1  protected byte[] readMsg(DataInputStream in ) throws IOException {
2  byte[] ret = null;
3
4  byte[] hdr = new byte[4];
5  byte[] temp = null;
6  boolean endHdr = false;
7  int r = 0;
8  int length = 0;
9  boolean endMsg = false;
10 int offset = 0;
11
12 while (!endMsg) {
13     try {
14         if (endHdr) {
15                 r = in.read(temp, offset, 1);
16         } else {
17                 r = in.read(hdr, offset, 1);
18         }
19     } catch (IOException e) {
20         log.info("Mistake reading data: " + e.getMessage());
```

```
21          throw e;
22      } catch (Exception e) {
23          log.info("readMsg Oops: " + e.getMessage());
24          throw new IOException();
25      }
26
27      if (r > 0) {
28          if (offset == 2) {
29              length = ((int) hdr[offset] & 0xFF) << 8;
30          }
31          if (offset == 3) {
32              length = length | (((int) hdr[offset] & 0xFF));
33              temp = new byte[length];
34              endHdr = true;
35              System.arraycopy(hdr, 0, temp, 0, 4);
36          }
37          if ((length > 0) && (offset == length - 1)) {
38              endMsg = true;
39          }
40          offset++;
41      } else if (r == -1) {
42          log.info("End of stream has been reached from " + this.remotePeerIP);
43          throw new IOException();
44      }
45  }
46  if (length > 0) {
47      ret = new byte[length];
48      System.arraycopy(temp, 0, ret, 0, length);
49  }
50  return ret;
51  }
```

```
1   protected byte[] readEncMsg(DataInputStream in, Cipher cipher) throws
        IOException {
2       byte[] decryptedMsg=null;
3       byte[] encryptedMsg=null;
4       ByteArrayOutputStream baos = new ByteArrayOutputStream();
5       CipherInputStream cis = new CipherInputStream(in, cipher);
6       byte[] buffer = new byte[1024];
7       int bytesRead;
8       while ((bytesRead = cis.read(buffer)) != -1) {
9           baos.write(buffer, 0, bytesRead);
10      }
11      encryptedMsg = baos.toByteArray();
12      System.out.println(new String(encryptedMsg,StandardCharsets.UTF_8));
13
14      // Decifra il messaggio
15      try {
16      decryptedMsg = cipher.doFinal(encryptedMsg);
17      } catch (IllegalBlockSizeException e) {
18         log.info("Error decrypting message due to illegal block size: " + e.
        getMessage());
19         throw new IOException();
20      } catch (BadPaddingException e) {
21         log.info("Error decrypting message due to bad padding: " + e.getMessage
        ());
22         throw new IOException();
23      }
```

```
24      // leggi il messaggio decifrato come se fosse un altro DataInputStream
    byte[] ret = null;
25      DataInputStream decryptedIn = new DataInputStream(new ByteArrayInputStream
    (decryptedMsg));
26      byte[] hdr = new byte[4];
27      byte[] temp = null;
28      boolean endHdr = false;
29      int r = 0;
30      int length = 0;
31      boolean endMsg = false;
32      int offset = 0;
33
34      while (!endMsg) {
35          try {
36              if (endHdr) {
37                      r = decryptedIn.read(temp, offset, 1);
38              } else {
39                      r = decryptedIn.read(hdr, offset, 1);
40              }
41          } catch (IOException e) {
42              log.info("Mistake reading data: " + e.getMessage());
43              throw e;
44          } catch (Exception e) {
45              log.info("readMsg Oops: " + e.getMessage());
46              throw new IOException();
47          }
48
49          if (r > 0) {
50              if (offset == 2) {
51                  length = ((int) hdr[offset] & 0xFF) << 8;
52              }
53              if (offset == 3) {
54                  length = length | (((int) hdr[offset] & 0xFF));
55                  temp = new byte[length];
56                  endHdr = true;
57                  System.arraycopy(hdr, 0, temp, 0, 4);
58              }
59              if ((length > 0) && (offset == length - 1)) {
60                  endMsg = true;
61              }
62              offset++;
63          } else if (r == -1) {
64              log.info("End of stream has been reached from " + this.
    remotePeerIP);
65              throw new IOException();
66          }
67      }
68      if (length > 0) {
69          ret = new byte[length];
70          System.arraycopy(temp, 0, ret, 0, length);
71      }
72      return ret;
73      }
```

```
1   protected byte[] readMsgEnc(CipherInputStream in ) throws IOException {
2
3   ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
4
5   byte[] buffer = new byte[1024];
```

```
int bytesRead;
while ((bytesRead = in.read(buffer)) != -1) {
outputStream.write(buffer, 0, bytesRead);
}

byte[] decryptedData = outputStream.toByteArray();
 ByteArrayInputStream bais = new ByteArrayInputStream(decryptedData);
DataInputStream dis = new DataInputStream(bais);
byte[] ret = null;

byte[] hdr = new byte[4];
byte[] temp = null;
boolean endHdr = false;
int r = 0;
int length = 0;
boolean endMsg = false;
int offset = 0;

while (!endMsg) {
try {
    if (endHdr) {
           r = dis.read(temp, offset, 1);
           //log.info("TEST PRINT after end hdr: "+new String(temp,
    StandardCharsets.UTF_8));
    } else {
           r = dis.read(hdr, offset, 1);
           //log.info("TEST PRINT hdr: "+new String(hdr,StandardCharsets.UTF_8
    ));
    }
} catch (IOException e) {
    log.info("Mistake reading data: " + e.getMessage());
    throw e;
} catch (Exception e) {
    log.info("readMsg Oops: " + e.getMessage());
    throw new IOException();
}

if (r > 0) {
    if (offset == 2) {
        length = ((int) hdr[offset] & 0xFF) << 8;
    }
    if (offset == 3) {
        length = length | (((int) hdr[offset] & 0xFF));
        temp = new byte[length];
        endHdr = true;
        System.arraycopy(hdr, 0, temp, 0, 4);
        log.info("TEST PRINT after end hdr: "+new String(temp,StandardCharsets
    .UTF_8));
    }
    if ((length > 0) && (offset == length - 1)) {
        endMsg = true;
    }
    offset++;
} else if (r == -1) {
    log.info("End of stream has been reached from " + this.remotePeerIP);
    throw new IOException();
}
}
if (length > 0) {
    ret = new byte[length];
    System.arraycopy(temp, 0, ret, 0, length);
```

```
64 }
65 return ret;
66 }
67
```

For the ReadEncMsg function, since we still used a DataOutputStream object associated with a CipherOutputStream that contains a Cipher object with the encrypting algorithm, the initialization vector and the session key, all the written output data are automatically enciphered. For the input, we used also the DataInputStream object associated with a CipherInputStream that contains a Cipher object with the encrypting algorithm, the initialization vector and the session key, so the input data read from the SSLSocket are automatically decrypted without other instruction. As a consequence, the standard readMsg function can be used and the readEncMsg function is used only when the CipherInputStream is disabled or not supported.

Chapter 7

Simulation and Testing

The simulation is done building a virtual machine that uses a Unix environment (Debian Bullseye OS) with the Vagrant tool; we built two virtual setups with the same topology described in chapter 4: the first virtual setup is with the two PCEs without PCEPS support, the second one instead has full PCEPS support for the PCEs .
Since we could not change the FRR code of the routers, the routers cannot speak PCEPS, so they will only use PCEP in our setup. That is why we focused on the analysis of the communication between the two PCEs.
First, we see if the changes to implement the new secure protocol(PCEPS) are working and how the PCEs now behave compared to the original Netphony-PCEs; furthermore we check if the two PCEs respect the packetflow and initialization rules of PCEPS how described before; lastly, we see if the encryption works and the performances for the setup with encryption and authentication enabled are still acceptable.

7.1 Setups Testing

In the following figures we have the packets flow and full output of the setup simulation with PCEPS disabled, so as we described in the previous sections, with the value PCEP_TLS inside the PCEPValues.java file set to zero:

Here we reported the full output of the PCE1 in domain1 and PCE2 in domain2 but with the value PCEP_TLS set to 1:

As we can see in figures 7.1, 7.2, 7.3, if we look at Thread 13 PCE1 first waits for a connection and then, when PCE2 starts running and opens a tcp channel with PCE1, it enters in PCEP_STATE_STARTTLS_WAIT and keeps waiting for a *StartTLS* message from PCE2 after the start of the STARTTLS wait timer; in the meanwhile it sends a *StartTLS* message to PCE2 in order to notify its intention of

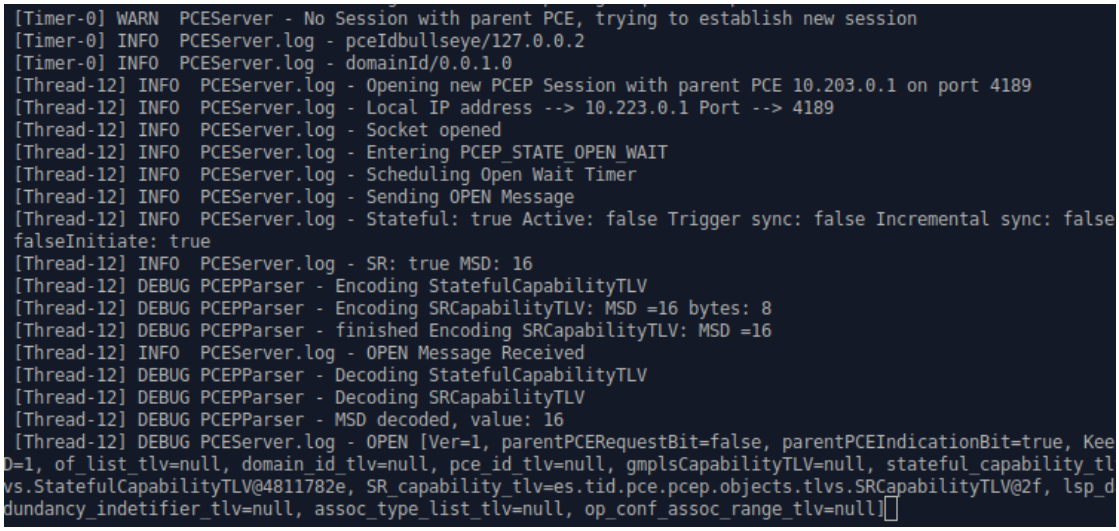

Figure 7.1: PCE1 in domain1 establishes a PCEP session with PCE2 in domain2

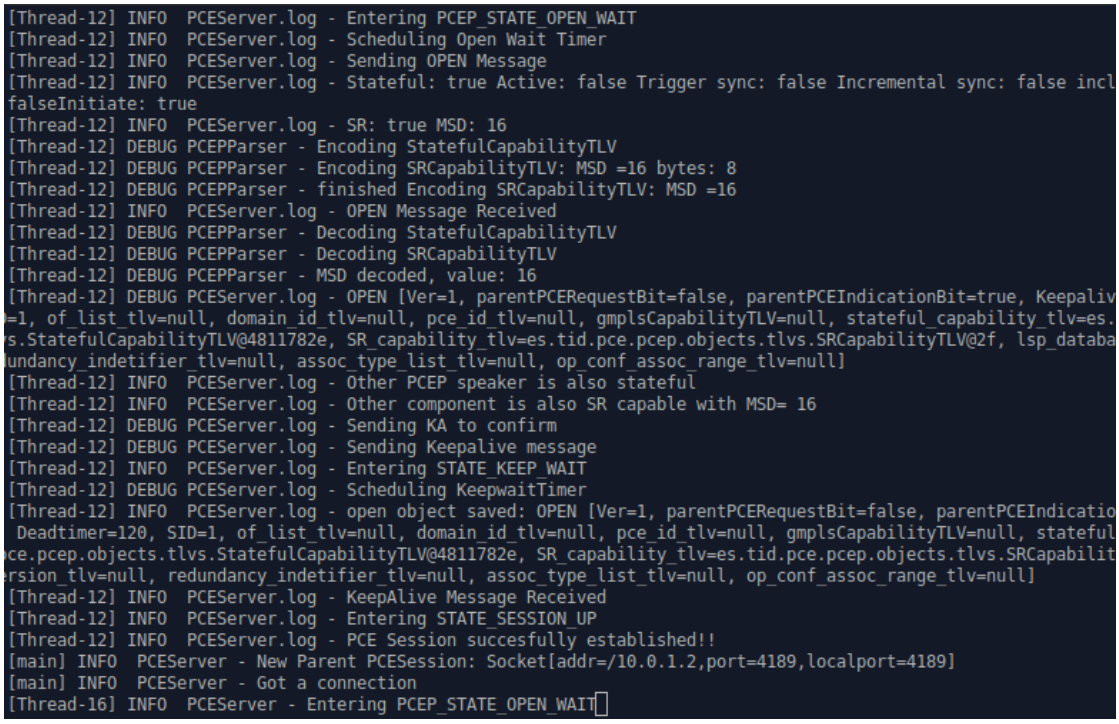

Figure 7.2: PCE1 in domain1 establishes a PCEP session with PCE2 in domain2

opening a secure channel with TLS.
When PCE2 *StartTLS* message is received from PCE1, it stops the STARTTLS wait timer and opens a secure TLS channel with parameters and algorithms that we

```
[Thread-18] DEBUG PCEServer - Reseting Dead Timer
[Thread-14] DEBUG PCEServer - Sending Keepalive message
[Thread-12] DEBUG PCEServer.log - KEEPALIVE message received
[Thread-12] INFO  PCEServer.log - Reseting Dead Timer as PCEP Message has arrived
[Thread-13] DEBUG PCEServer - Reseting Dead Timer
[Thread-20] DEBUG PCEServer - PCEP KEEPALIVE message received from /10.203.0.1
[Thread-20] DEBUG PCEServer - Reseting Dead Timer as PCEP Message has arrived in /10.203.0.1
[Thread-22] DEBUG PCEServer - Reseting Dead Timer
[Thread-23] DEBUG PCEServer - Sending Keepalive message
[Thread-20] DEBUG PCEServer - PCEP KEEPALIVE message received from /10.203.0.1
[Thread-20] DEBUG PCEServer - Reseting Dead Timer as PCEP Message has arrived in /10.203.0.1
[Thread-12] DEBUG PCEServer.log - KEEPALIVE message received
[Thread-12] INFO  PCEServer.log - Reseting Dead Timer as PCEP Message has arrived
[Thread-22] DEBUG PCEServer - Reseting Dead Timer
[Thread-14] DEBUG PCEServer - Sending Keepalive message
[Thread-13] DEBUG PCEServer - Reseting Dead Timer
[Timer-0] WARN  PCEServer - There is a session with Parent PCE!
[Thread-16] DEBUG PCEServer - PCEP KEEPALIVE message received from /10.0.1.2
[Thread-16] DEBUG PCEServer - Reseting Dead Timer as PCEP Message has arrived in /10.0.1.2
[Thread-16] DEBUG PCEServer - PCEP KEEPALIVE message received from /10.0.1.2
[Thread-16] DEBUG PCEServer - Reseting Dead Timer as PCEP Message has arrived in /10.0.1.2
[Thread-18] DEBUG PCEServer - Reseting Dead Timer
[Thread-19] DEBUG PCEServer - Sending Keepalive message
[Thread-12] DEBUG PCEServer.log - KEEPALIVE message received
[Thread-12] INFO  PCEServer.log - Reseting Dead Timer as PCEP Message has arrived
[Thread-13] DEBUG PCEServer - Reseting Dead Timer
[Thread-20] DEBUG PCEServer - PCEP KEEPALIVE message received from /10.203.0.1
[Thread-20] DEBUG PCEServer - Reseting Dead Timer as PCEP Message has arrived in /10.203.0.1
[Thread-22] DEBUG PCEServer - Reseting Dead Timer
[Thread-16] DEBUG PCEServer - PCEP KEEPALIVE message received from /10.0.1.2
[Thread-16] DEBUG PCEServer - Reseting Dead Timer as PCEP Message has arrived in /10.0.1.2
[Thread-18] DEBUG PCEServer - Reseting Dead Timer
```

Figure 7.3: PCE1 in domain1 establishes a PCEP session with PCE2 in domain2

```
[main] INFO  PCEServer - New Parent PCESession: Socket[addr=/10.223.0.1,port=48353,localport=4189]
[main] INFO  PCEServer - Got a connection
[Thread-13] INFO  PCEServer - Entering PCEP_STATE_STARTTLS_WAIT
[Thread-13] INFO  PCEServer - Scheduling STARTTLS Wait Timer
[Thread-13] INFO  PCEServer - Sending STARTTLS msg
[Thread-13] INFO  PCEServer - STARTTLS message received
[Thread-13] DEBUG PCEServer - es.tid.pce.pcep.messages.PCEPStarttls@6deddd39
[Thread-13] INFO  PCEServer - SECURE SOCKET OPENED WITH TLS 1.3
[Thread-13] INFO  PCEServer - Entering PCEP_STATE_OPEN_WAIT
[Thread-13] INFO  PCEServer - Scheduling Open Wait Timer
[Thread-13] INFO  PCEServer - Sending OPEN Message
[Thread-13] INFO  PCEServer - Stateful: true Active: false Trigger sync: false Incremental sync: false in
eInitiate: true
[Thread-13] INFO  PCEServer - SR: true MSD: 16
[Thread-13] DEBUG PCEPParser - Encoding StatefulCapabilityTLV
```

Figure 7.4: PCE1 in domain1 establishes a PCEPS session with PCE2 in domain2

described in the code showed in the previous sections (we can see that everything until now works as intended from the log message displayed that says SECURE SOCKET OPENED WITH TLS 1.3).
At this point the initialization of the secure channel between the two peers is done correctly, so they are both now speaking PCEP over TLS; now the initialization phase of a PCEP starts, so as we can see PCE1 enters in Open wait state, schedule

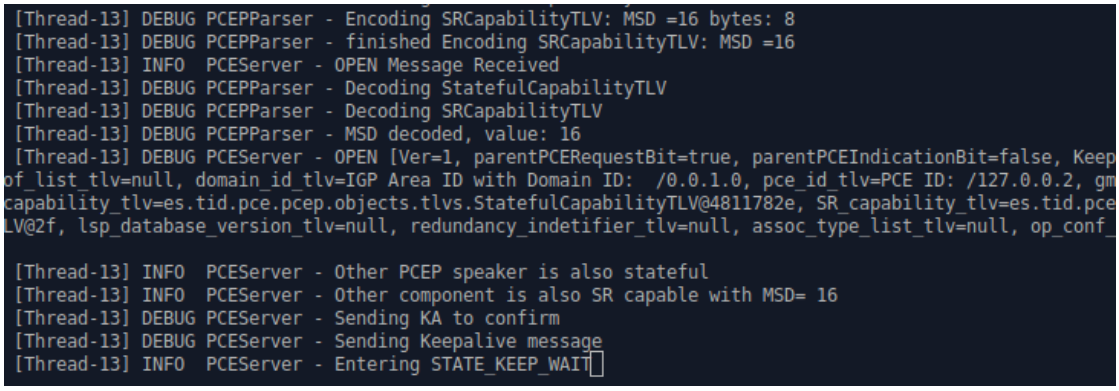

Figure 7.5: PCE1 in domain1 establishes a PCEPS session with PCE2 in domain2

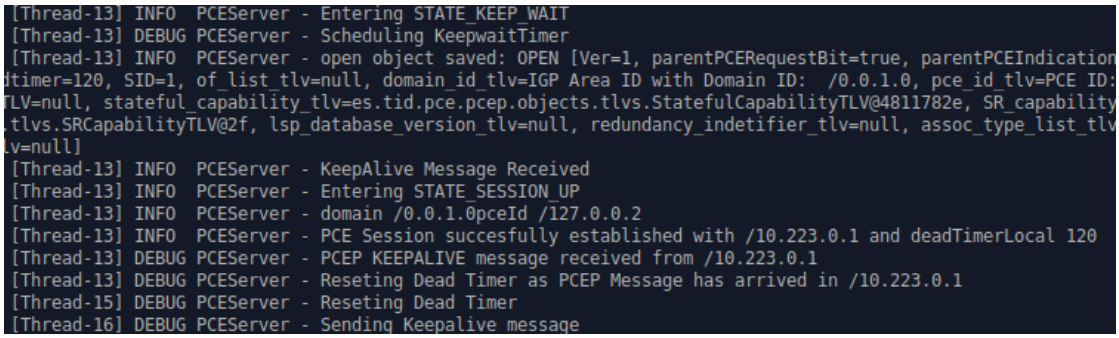

Figure 7.6: PCE1 in domain1 establishes a PCEPS session with PCE2 in domain2

the *OpenWait* timer correctly and sends to PCE2 an *Open* message with all the correct parameter inside. After a while PCE1 receives from PCE2 an *Open* message, it decodes it correctly and from this message, it sets all the session parameters accordingly. Now the PCE1 has officially established successfully the PCEPS session and enters in the KeepWait state.
In this state the two peers (PCE1 and PCE2) send to each other *Keepalive* messages to keep up the connection. From here on now the communication between the two peers can be considered secure since its all over TLS.

Figures 7.7, 7.8, 7.9 show the output of the simulation from PCE2 point of view with the packets flow of the initialization phase, that has the same behaviour as the PCE1 packets flow that was described and shown before.
With this setup confidentiality of the packets (in this case the *Open* packets and all their parameters) is guaranteed, but also the integrity of the packets, since AES in GCM mode also assures this property. The simulation was configured once with the TLS client authentication disabled (so only the server authentication was enabled), then with client authentication enabled in order to see how the setup

```
[Thread-12] INFO  PCEServer.log - Opening new PCEP Session with parent PCE 10.203.0.1 on port 4189
[Thread-12] INFO  PCEServer.log - Local IP address --> 10.223.0.1 Port --> 4189
[Thread-12] INFO  PCEServer.log - Socket opened
```

Figure 7.7: PCE2 in domain2 establishes a PCEPS session with PCE1 in domain1

```
[Thread-12] INFO  PCEServer.log - Local IP address --> 10.223.0.1 Port --> 4189
[Thread-12] INFO  PCEServer.log - Socket opened
[Thread-12] INFO  PCEServer.log - Entering PCEP_STATE_STARTTLS_WAIT
[Thread-12] INFO  PCEServer.log - Scheduling STARTTLS Wait Timer
[Thread-12] INFO  PCEServer.log - Sending STARTTLS msg
[Thread-12] INFO  PCEServer.log - STARTTLS message received
[Thread-12] DEBUG PCEServer.log - es.tid.pce.pcep.messages.PCEPStarttls@6deddd39
[Thread-12] INFO  PCEServer.log - SECURE SOCKET OPENED WITH TLS 1.3
[Thread-12] INFO  PCEServer.log - Entering PCEP_STATE_OPEN_WAIT
[Thread-12] INFO  PCEServer.log - Scheduling Open Wait Timer
[Thread-12] INFO  PCEServer.log - Sending OPEN Message
[Thread-12] INFO  PCEServer.log - Stateful: true Active: false Trigger sync: false Incremental sync: false incl
falseInitiate: true
[Thread-12] INFO  PCEServer.log - SR: true MSD: 16
[Thread-12] DEBUG PCEPParser - Encoding StatefulCapabilityTLV
[Thread-12] DEBUG PCEPParser - Encoding SRCapabilityTLV: MSD =16 bytes: 8
[Thread-12] DEBUG PCEPParser - finished Encoding SRCapabilityTLV: MSD =16
[Thread-12] INFO  PCEServer.log - OPEN Message Received
[Thread-12] DEBUG PCEPParser - Decoding StatefulCapabilityTLV
[Thread-12] DEBUG PCEPParser - Decoding SRCapabilityTLV
[Thread-12] DEBUG PCEPParser - MSD decoded, value: 16
[Thread-12] DEBUG PCEServer.log - OPEN [Ver=1, parentPCERequestBit=false, parentPCEIndicationBit=true, Keepalive
)=1, of_list_tlv=null, domain_id_tlv=null, pce_id_tlv=null, gmplsCapabilityTLV=null, stateful_capability_tlv=es.t
vs.StatefulCapabilityTLV@4811782e, SR_capability_tlv=es.tid.pce.pcep.objects.tlvs.SRCapabilityTLV@2f, lsp_databas
dundancy_indetifier_tlv=null, assoc_type_list_tlv=null, op_conf_assoc_range_tlv=null]
[Thread-12] INFO  PCEServer.log - Other PCEP speaker is also stateful
[Thread-12] INFO  PCEServer.log - Other component is also SR capable with MSD= 16
[Thread-12] DEBUG PCEServer.log - Sending KA to confirm
[Thread-12] DEBUG PCEServer.log - Sending Keepalive message
[Thread-12] INFO  PCEServer.log - Entering STATE_KEEP_WAIT
[Thread-12] DEBUG PCEServer.log - Scheduling KeepwaitTimer
[Thread-12] INFO  PCEServer.log - open object saved: OPEN [Ver=1, parentPCERequestBit=false, parentPCEIndicatio
 Deadtimer=120, SID=1, of_list_tlv=null, domain_id_tlv=null, pce_id_tlv=null, gmplsCapabilityTLV=null, stateful
```

Figure 7.8: PCE2 in domain2 establishes a PCEPS session with PCE1 in domain1

performances could change.
Inside the simulation we could also see the packets using the sniffer tool Tshark on PCE1-PCE2 virtual link in order to capture the TLS packets and PCEP messages: we could see notice that *Open* packets and *Keepalive* packets were all well encrypted (in particular the *Open* packets, which contain sensitive data for the PCEPS session) and the TLS handshake with peer authentication is done correctly.
From a performance point of view, we could not see any meaningful changes comparing the PCEPS enabled setup and standard PCEP setup, even when client authentication was enabled. Still it is important to notice that we didn't use any key exchange algorithm for the session key of PCE1 and PCE2 because we simply hard coded the session key inside the Netphony-PCE modified source code, just as an academic demonstration. However it is unluckily that using a key exchange algorithm could have a significant impact on the performances.

```
[Thread-12] INFO  PCEServer.log - open object saved: OPEN [Ver=1, parentPCERequestBit=false, parentPCEIndica
, Deadtimer=120, SID=1, of_list_tlv=null, domain_id_tlv=null, pce_id_tlv=null, gmplsCapabilityTLV=null, state
pce.pcep.objects.tlvs.StatefulCapabilityTLV@4811782e, SR_capability_tlv=es.tid.pce.pcep.objects.tlvs.SRCapabi
ersion_tlv=null, redundancy_indetifier_tlv=null, assoc_type_list_tlv=null, op_conf_assoc_range_tlv=null]
[Thread-12] INFO  PCEServer.log - KeepAlive Message Received
[Thread-12] INFO  PCEServer.log - Entering STATE_SESSION_UP
[Thread-12] INFO  PCEServer.log - PCE Session succesfully established!!
[Thread-14] DEBUG PCEServer - Sending Keepalive message
[Thread-12] DEBUG PCEServer.log - KEEPALIVE message received
[Thread-12] INFO  PCEServer.log - Reseting Dead Timer as PCEP Message has arrived
[Thread-13] DEBUG PCEServer - Reseting Dead Timer
[Thread-14] DEBUG PCEServer - Sending Keepalive message
[Thread-12] DEBUG PCEServer.log - KEEPALIVE message received
[Thread-12] INFO  PCEServer.log - Reseting Dead Timer as PCEP Message has arrived
[Thread-13] DEBUG PCEServer - Reseting Dead Timer
[Thread-14] DEBUG PCEServer - Sending Keepalive message
[Thread-12] DEBUG PCEServer.log - KEEPALIVE message received
[Thread-12] INFO  PCEServer.log - Reseting Dead Timer as PCEP Message has arrived
```

Figure 7.9: PCE2 in domain2 establishes a PCEPS session with PCE1 in domain1

7.2 Discussion

In this thesis we have looked at the practical feasibility of deploying a multi-domain Path Computation Element (PCE) setup that is scalable and secure, in order to guarantee security properties that are needed to be used as the Path Controller part of the UPIN framework.
The research focus was to analyse the PCE Communication protocol, find the vulnerabilities, weaknesses, the actual state of the art technology and in the end find a valid implementation which can be used inside the setup already used to develop the UPIN framework. Our proof of concept shows the successful provisioning of the computation and installation of a multi-domain switch path (LSP) using a secure and reliable communication.
The first roadblock in this research was that currently, almost all commercial deployments do not support any security features for PCE based communication, and the only one that does it is an incomplete secure implementation and not open-source. We have shown our evaluation of the available PCEs in chapter 3.
In order to make future researches easier, we chose a lightweight and open-source router stack on top of Linux, the Free Range Routing (FRR), which support PCEP protocol and its possible new implementations. Note that this setup has some limitations: first of all, the Free Range Routing (FRR) PCEP implementation is still experimental and not completely released while working on this thesis; second, the Multi-Protocol Label Switching (MPLS) implementation of the linux-kernel has some limitations, for instance the kernel can pop at most one label. To forward the packets from the Area Border Router (ABR) toward the other domain, a php script from a previous work was used. All of this can be a problem in different use cases regarding the Segment Identifiers. However this problem were out of the scope of this work but it is important to address them in future works.

Concerning the PCEP protocol, there is still the Transition problem: at the moment of this work there are no devices that support PCEPS, so like in our setup with PCEPS implemented in TLS, you can choose to allow also standard PCEPS sessions with the routers which cannot establish a PCEPS connection until every node in the network is upgraded or you can put a limitation and strictly allow only secure connection and block any attempt of standard PCEP session establishment from node that are still not updated. With the first option, it is easier to manage the network even if old devices are present, however it is prone to malicious attacks that can disrupt the network.
It is also possible to at least make an implementation of a PCE without TLS and all the security features of PCEPS (so without the initialization and new packet flow present in PCEPS) but with the use of cryptographic algorithm in order to have a small set of security features guaranteed. Also in this case note that for the key exchange algorithms and other cryptographic features, a key pair and a certificate could be needed and have them is not always the case for every devices.

Chapter 8

Conclusion

We aimed in this research to answer the question of how we can perform multi-domain segment routing in the Path Controller part of the UPIN framework, which is build upon PCE devices that rely on an insecure protocol, that is the PCEP protocol, developing a new version of it more secure and reliable. Also we asked ourselves which was the current state of the art technologies.
We explored the inner protocol insecurities, analysed its behaviour and evaluated different possibilities to improve it. Furthermore, we also discussed the Transition problem that occurs when the upgraded PCEs and PCEPS protocol are introduced in an old network setup and what are the possible solutions to deal with it; we also presented the new possible attacks against a new version of the Path Controller that makes use of these new technologies.
While building the proof of concept we ran on multiple limitations that were unknown beforehand. Currently there are no PCE implementations that take in consideration security features, and the only one that has at least a TLS implementation is not open-source, so it is not possible to modify for our own needs. Also the current support and implementation of the PCEP protocol in most of the routers are still not enough mature to deal in an efficient way with the Transition problem, so further security layers are needed to take care of the remaining attack vectors; lastly the guidelines for PCEP over TLS it is only in its draft phase.
The main contribution of this research is showing which are the insecurities of PCEP, which could be its improvements and to present a possible implementation that also works in a multi-domain environment.
In the end, with this work we achieve a new improvement for a crucial part of the UPIN framework, making the Path Controller more secure and reliable.

8.1 Future Works

For future works, it is possible to search a way to implement a valid key exchange algorithm and improve the PCEPS part concerning when it is really needed the client authentication and when it is not and evaluate the level of performances; Moreover it is possible using our setup to evaluate exactly which could be the data that must be protected and which could be left unprotected in order to treat efficiently the Transition problem deciding when to use the new technologies and when it is possible to still use the old one.

Bibliography

[1] JP Vasseur, Adrian Farrel, and Gerald Ash. *A Path Computation Element (PCE)-Based Architecture*. RFC 4655. Aug. 2006. DOI: 10.17487/RFC4655. URL: https://www.rfc-editor.org/info/rfc4655 (cit. on p. 1).

[2] Cees Portegies, Marijke Kaat, and Paola Grosso. «Supporting VNF chains: an implementation using Segment Routing and PCEP». In: *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 2021, pp. 1–5. DOI: 10.1109/ICIN51074.2021.9385527 (cit. on pp. 2, 10).

[3] Rodrigo Bazo, Cristian E.W. Hesselman, Leonardo Boldrini, and Paola Grosso. «Increasing the Transparency, Accountability and Controllability of multi-domain networks with the UPIN framework». English. In: *TAURIN'21: Proceedings of the ACM SIGCOMM 2021 Workshop on Technologies, Applications, and Uses of a Responsible Internet*. ACM SIGCOMM Workshop on Technologies, Applications, and Uses of a Responsible Internet, TAURIN 2021, TAURIN 2021 ; Conference date: 23-08-2021 Through 23-08-2021. Aug. 2021, pp. 8–13. DOI: 10.1145/3472951.3473506 (cit. on p. 2).

[4] Leonardo Boldrini, Rodrigo Bazo, Cristian E.W. Hesselman, and Paola Grosso. «UPIN - A shift in network control from operator to end user». English. In: ICT Open 2021, ICT.OPEN2021 ; Conference date: 10-02-2021 Through 11-02-2021. 2021. URL: https://www.ictopen.nl/ (cit. on p. 2).

[5] Holz R. et al. Hesselman C. Grosso P. «A Responsible Internet to Increase Trust in the Digital World». In: *Journal of Network and Systems Management* 28 (Oct. 2020), pp. 882–922. DOI: https://doi.org/10.1007/s10922-020-09564-7. URL: https://doi.org/10.1007/s10922-020-09564-7 (cit. on pp. 2, 5).

[6] Tomi Dufva and Mikko Dufva. «Grasping the future of the digital society». In: *Futures* 107 (2019), pp. 17–28. ISSN: 0016-3287. DOI: https://doi.org/10.1016/j.futures.2018.11.001. URL: https://www.sciencedirect.com/science/article/pii/S0016328717302252 (cit. on p. 2).

[7] Diego Lopez, Oscar Gonzalez de Dios, Qin Wu, and Dhruv Dhody. *PCEPS: Usage of TLS to Provide a Secure Transport for the Path Computation Element Communication Protocol (PCEP)*. RFC 8253. Oct. 2017. DOI: 10.17487/RFC8253. URL: https://www.rfc-editor.org/info/rfc8253 (cit. on pp. 3, 38).

[8] Leonardo Boldrini, Matteo Bachiddu, and Paola Grosso. «Implications of using PCEPS in PCE-based multi-domain networks». In: *CompSys 2022, Dutch Computer Science Conference, 8-10 june 2022, Kasteel de Vanenburg*. 2022 (cit. on p. 4).

[9] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. *Segment Routing Architecture*. RFC 8402. July 2018. DOI: 10.17487/RFC8402. URL: https://www.rfc-editor.org/info/rfc8402 (cit. on p. 10).

[10] Ahmed Bashandy, Clarence Filsfils, Stefano Previdi, Bruno Decraene, and Stephane Litkowski. *Segment Routing MPLS Interworking with LDP*. RFC 8661. Dec. 2019. DOI: 10.17487/RFC8661. URL: https://www.rfc-editor.org/info/rfc8661 (cit. on p. 11).

[11] Ahmed Bashandy, Clarence Filsfils, Stefano Previdi, Bruno Decraene, Stephane Litkowski, and Rob Shakir. *Segment Routing with the MPLS Data Plane*. RFC 8660. Dec. 2019. DOI: 10.17487/RFC8660. URL: https://www.rfc-editor.org/info/rfc8660 (cit. on p. 11).

[12] JP Vasseur and Jean-Louis Le Roux. *Path Computation Element (PCE) Communication Protocol (PCEP)*. RFC 5440. Mar. 2009. DOI: 10.17487/RFC5440. URL: https://www.rfc-editor.org/info/rfc5440 (cit. on p. 13).

[13] Andy Heffernan. *Protection of BGP Sessions via the TCP MD5 Signature Option*. RFC 2385. Aug. 1998. DOI: 10.17487/RFC2385. URL: https://www.rfc-editor.org/info/rfc2385 (cit. on p. 26).

[14] Dr. Joseph D. Touch, Ron Bonica, and Allison J. Mankin. *The TCP Authentication Option*. RFC 5925. June 2010. DOI: 10.17487/RFC5925. URL: https://www.rfc-editor.org/info/rfc5925 (cit. on p. 26).

[15] JP Vasseur, Jean-Louis Le Roux, Raymond Zhang, and Dr. Nabil N. Bitar. *A Backward-Recursive PCE-Based Computation (BRPC) Procedure to Compute Shortest Constrained Inter-Domain Traffic Engineering Label Switched Paths*. RFC 5441. Apr. 2009. DOI: 10.17487/RFC5441. URL: https://www.rfc-editor.org/info/rfc5441 (cit. on pp. 31, 32).

[16] Olivier Dugeon, Julien Meuric, Young Lee, and Daniele Ceccarelli. *PCEP Extension for Stateful Inter-Domain Tunnels*. Internet-Draft draft-ietf-pce-stateful-interdomain-03. Work in Progress. Internet Engineering Task Force, Mar. 2022. 37 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-pce-stateful-interdomain/03/ (cit. on p. 32).

[17] Siva Sivabalan, Clarence Filsfils, Jeff Tantsura, Wim Henderickx, and Jonathan Hardwick. *Path Computation Element Communication Protocol (PCEP) Extensions for Segment Routing*. RFC 8664. Dec. 2019. DOI: 10.17487/RFC8664. URL: https://www.rfc-editor.org/info/rfc8664 (cit. on p. 33).

[18] Fatai Zhang, Quintin Zhao, Oscar Gonzalez de Dios, R. Casellas, and Daniel King. *Path Computation Element Communication Protocol (PCEP) Extensions for the Hierarchical Path Computation Element (H-PCE) Architecture*. RFC 8685. Dec. 2019. DOI: 10.17487/RFC8685. URL: https://www.rfc-editor.org/info/rfc8685 (cit. on p. 33).

[19] Mahesh Jethanandani, Keyur Patel, and Lianshu Zheng. *Analysis of BGP, LDP, PCEP, and MSDP Issues According to the Keying and Authentication for Routing Protocols (KARP) Design Guide*. RFC 6952. May 2013. DOI: 10.17487/RFC6952. URL: https://www.rfc-editor.org/info/rfc6952 (cit. on p. 40).

[20] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. RFC 7457. Feb. 2015. DOI: 10.17487/RFC7457. URL: https://www.rfc-editor.org/info/rfc7457 (cit. on p. 49).

[21] Bodo Moeller and Adam Langley. *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*. RFC 7507. Apr. 2015. DOI: 10.17487/RFC7507. URL: https://www.rfc-editor.org/info/rfc7507 (cit. on p. 49).

[22] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. *Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7525. May 2015. DOI: 10.17487/RFC7525. URL: https://www.rfc-editor.org/info/rfc7525 (cit. on p. 50).