UNIVERSIDADE D
COIMBRA

Maria Carolina Jordão Pereira Gonçalves

Authentication and accounting framework for SDN controller

Dissertation in the context of the Master in Informatics Engineering, Specialization in Communications, Services, and Infrastructures, advised by Professor Dr. Bruno Sousa and by Professor Dr. Nuno Antunes and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

July 2022

Maria Carolina Jordão Pereira Gonçalves
Authentication and accounting framework for SDN controller
UNIVERSIDADE D COIMBRA

Faculty of Sciences and Technology
Department of Informatics Engineering

Authentication and accounting framework for SDN controller

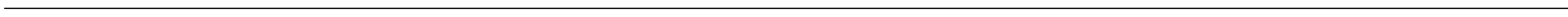
Maria Carolina Jordão Pereira Gonçalves

Dissertation in the context of the Master in Informatics Engineering, Specialization in Communications Services and Infrastructures advised by Prof. Dr. Bruno Sousa and Prof. Dr. Nuno Antunes and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

July 2022

UNIVERSIDADE Ð
COIMBRA

This page is intentionally left blank.



This document is written in American English

This page is intentionally left blank.

This work falls in the communications, services, and infrastructure specialization area and it was partially carried out in the Laboratory of Communications and Telematics (LCT) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work is partially supported by the project Adaptive, Intelligent and Distributed Assurance Platform (AIDA) (POCI-01-0247-FEDER-045907), co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation - COMPETE 2020 and by the Portuguese Foundation for Science and Technology (FCT), under CMU Portugal.

It is also partially supported by the project METRICS (POCI-01-0145-FEDER-032504), co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the the European Regional Development Fund (ERDF) through Portugal 2020 - Operacional Program for Competitiveness and Internationalisation.

This work has been supervised by Professor Bruno Sousa and Professor Nuno Antunes, both Assistant Professors at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

This page is intentionally left blank.

Acknowledgements

First, I would like to express my gratitude to my advisors Professor Bruno Sousa and Professor Nuno Antunes who were always there to guide me throughout this work. Their insight, wisdom, and guidance were always appreciated and crucial in given moments. I made some little mistakes, but their patience and willingness to teach me were always present, which resulted in me learning so much with this work.

In addition, I would like to thank my friends and colleagues, especially for all the spontaneous small breaks, fun, and enjoyable lunches, the 5 minutes talk that transformed into 30 minutes or an hour of conversation, the silly videos and music we watched and listened together and so much more.

At last, I would like to thank my parents for all the love and support I ever received. They shaped me into the person I am today, a determined, ambitious, independent, and caring person. I would also like to thank my sister, who was always there guiding me throughout all stages of my life, especially guiding me as a young adult, whether in personal or academic matters.

This page is intentionally left blank.

Abstract

These days, network administration has become easier because of advances in technologies such as Software Defined Networking (SDN), which offers more flexibility, therefore justifying the increase of its usage in cloud computing. However, SDN showed to be vulnerable to different attacks, which is concerning since SDN centralizes network information. To improve user management and security, several SDN controllers began to add Authorization, Authentication, and Accounting (AAA) services, which include the integration of protocols such as OAuth 2.0. Nevertheless, the basic authentication used in these services (e.g. passwords) or the use of OAuth 2.0 with basic tokens, which are Base64-encoded username and password, are still extremely vulnerable to various exploits.

The main objective of this work is to design and evaluate a framework with authentication and accounting mechanisms in a SDN controller. We started by analyzing which SDN controller should be chosen for this integration - OpenDaylight, as well as the identification and authentication approach - OpenID Connect (OIDC). We introduced the concept of trust levels, through the context information of a user's device, provided in the form of OIDC claims and created an algorithm to verify the context information in the OpenID Provider (OP). An experimental procedure was done to evaluate the addition of context information in OIDC, which showed good results only presenting a small additional cost in the overall performance of a OIDC authentication flow.

To begin developing the framework suggested, an architecture for the framework was proposed and described, highlighting its key elements: the OpenDaylight controller, the AAA filter, the Relying Party (RP), and Keycloak. Along with the introduction of the RP component in the controller, we also discuss Keycloak setups and modifications made to the controller's AAA filter. Finally, we prepared an experimental setup to retrieve metrics to analyze the framework, during the authentication process, renewal of a token, verification of a token, and to analyze the security and functionality of the roles and context information in OpenDaylight. We also compared the performance of authenticating in the standard OpenDaylight and in the framework proposed. The results obtained showed **a small cost in the overall performance of authentication with OIDC and context information in OpenDaylight, but a substantial increase in security in the controller.**

Keywords

SDN Controller, OpenID Connect, OpenDaylight, Trust Context Information, AAA, Keycloak

This page is intentionally left blank.

Resumo

Hoje em dia, a administração de redes tem se tornado mais fácil devido aos avanços em tecnologias como Software Defined Networking (SDN), oferecendo mais flexibilidade nestas, o que justifica o aumento da sua utilização em computação *cloud*. No entanto, *SDN* mostrou-se vulnerável a diferentes tipos de ataques, o que é preocupante uma vez que um controlador *SDN* centraliza informação da rede. Para melhorar a gestão e segurança dos utilizadores , vários controladores *SDN* começaram a acrescentar serviços de Autorização, Autenticação e *Accouting* (AAA), que incluem a integração de protocolos como o OAuth 2.0. No entanto, a autenticação básica utilizada nestes serviços (ex: password) ou a utilização do OAuth 2.0 com *basic tokens*, que são *username* e palavra-passe codificados na Base64, continuam a ser extremamente vulneráveis a vários *exploits*.

O principal objectivo deste trabalho é planear e avaliar uma *framework* com mecanismos de autenticação e *accouting* num controlador *SDN*. Começámos por analisar qual o controlador *SDN* a ser escolhido para esta integração, *OpenDaylight*, bem como o mecanismo de autenticação, *OIDC*. Introduzimos também uma ideia de níveis de confiança, através da informação de contexto de um dispositivo do utilizador, cuja informação é acrescentada nas *claims* do *OIDC* e criámos um algoritmo para verificar a informação de contexto no dispositivo OP. Foi feito um procedimento experimental para avaliar a adição de informação de contexto em *OIDC*, que mostrou bons resultados apresentando apenas um pequeno custo no desempenho global de um fluxo de autenticação *OIDC*.

Para começar a desenvolver a *framework* sugerida, foi proposta e descrita uma arquitectura para a *framework*, destacando os seus elementos-chave: o controlador OpenDaylight, o filtro AAA, o Relying Party (RP), e o Keycloak. Juntamente com a introdução do componente RP no controlador, discutimos também as configurações do Keycloak e as modificações feitas no filtro AAA do controlador. Finalmente, foi preparado um procedimento experimental para obter métricas de modo a analisar a *framework*, durante o processo de autenticação, renovação de um token, verificação de um token, e a analisar a segurança e funcionalidade dos *roles* e informação de contexto no OpenDaylight. Também comparámos o desempenho da autenticação no OpenDaylight orginal e na *framework* proposta. Os resultados obtidos mostraram um **pequeno custo no desempenho global da autenticação com OIDC e informação de contexto no OpenDaylight, mas um aumento substancial da segurança no controlador.**

Palavras-Chave

Controlador SDN, OpenID Connect, OpenDaylight, Informação de Contexto de Confiança, AAA, Keycloak

Contents

1 Introduction **6**
1.1 Objectives . . . 8
1.2 Contributions . . . 8
1.3 Structure . . . 9

2 Background and Related Work **11**
2.1 Security Features . . . 11
2.1.1 OAuth 2.0 . . . 11
2.1.2 OpenID Connect . . . 14
2.1.3 SAML2.0 . . . 16
2.1.4 Authorization Authentication and Accounting . . . 17
2.2 Related Work . . . 18
2.3 Authentication and authorization tools . . . 20
2.3.1 Authlib . . . 20
2.3.2 Keycloak . . . 21
2.4 Software Defined Network Controllers . . . 23
2.4.1 Open Network Operating System . . . 24
2.4.2 OpenDaylight . . . 24
2.4.3 Lighty.io . . . 25
2.4.4 Open Network Automation Platform . . . 26
2.4.5 Ryu . . . 26
2.4.6 Comparison . . . 27
2.5 Software Defined Network & related protocols . . . 27
2.5.1 REST interfaces . . . 27
2.5.2 OpenFlow . . . 28
2.5.3 P4 . . . 28
2.5.4 NetConf . . . 28
2.6 Summary . . . 29

3 Research Objectives and Approach **31**
3.1 Research Methodology . . . 31
3.2 Research Objectives . . . 32
3.3 Approach . . . 33
3.3.1 Evaluate the impact of the additional parameters with context information in OIDC . . . 33
3.3.2 Enable the support of OIDC for authentication and authorization in SDN controllers . . . 35
3.3.3 Enable the support of OIDC with an extension for additional parameters with context information in an OpenDaylight controller . . . 37
3.4 Use case . . . 37
3.5 Architecture . . . 37

4 Preliminary Results **40**
4.1 Integration . . . 40
4.2 Evaluation . . . 42
4.2.1 Scenario . . . 42
4.2.2 Configuration parameters . . . 43
4.2.3 Metrics . . . 44
4.3 Results . . . 44
4.4 Summary . . . 46

5 Implementation **47**
5.1 Concepts . . . 47
5.1.1 Roles and Grants . . . 47
5.2 Components . . . 49
5.2.1 Keycloak . . . 49
5.2.2 Relying Party . . . 56
5.2.3 AAA filter . . . 57
5.3 Steps for Authentication and Authorization . . . 57
5.3.1 Authentication . . . 58
5.3.2 Verification of RPT token . . . 59
5.3.3 Renewal of RPT token . . . 62

6 Setup and Evaluation **63**
6.1 Experimental Setup . . . 63
6.1.1 Authentication Process . . . 64
6.1.2 Refresh/Renewing tokens process . . . 64
6.1.3 Standard authentication and OIDC authentication in OpenDaylight 67
6.1.4 Access denied . . . 69
6.2 Evaluation Results . . . 69
6.2.1 Authentication Process . . . 69
6.2.2 Refresh/Renewing tokens process . . . 74
6.2.3 Standard authentication and OIDC authentication in OpenDaylight 75
6.2.4 Access denied . . . 75
6.3 Enhanced security with OIDC . . . 76
6.3.1 Security with OIDC without context information . . . 77
6.3.2 Security with OIDC and trust context information . . . 77
6.3.3 Comparison of OpenDaylight security integrations . . . 78

7 Conclusion **82**
7.1 Future work . . . 83

A BIANFE: Object identification and authentication in federated scenarios **91**

This page is intentionally left blank.

Acronyms

AAA Authorization Authentication and Accounting. 6, 11, 17, 20, 21, 24–27, 29, 30, 32, 33, 38, 39, 47, 48, 57, 58

AES Advanced Encryption Standard. 22, 34

API Application Programming Interface. 6, 7, 12, 19, 23, 24, 26, 27, 29, 50, 56–58

AS Authorization Server. 12–14, 17

AUSF Authentication Server Function. 18, 29

CLI Command-line Interface. 6, 24, 38, 39

CRUD Create, Read, Update and Delete. 27

FIM Federated Identity Management. 8, 9, 20, 21

GUI Graphical User Interface. 24, 26, 35

HTTP Hypertext Transfer Protocol. 19, 27, 50, 51, 57, 60, 61, 67, 68, 75, 83

HTTPS Hypertext Transfer Protocol Secure. 9, 41

IdP Identity Provider. 16, 17, 20, 26

JWT JSON Web Token. 7, 14, 19, 26

OIDC OpenID Connect. i, ix, xi, xiii, xiv, xix, xx, 3, 6–9, 14–16, 19–23, 26, 29–35, 37, 38, 40–47, 49, 50, 56–58, 62–64, 67, 68, 71, 75–80, 82, 83

ONAP Open Network Automation Platform. 26, 30

ONOS Open Network Operating System. 19, 24, 27

OP OpenID Provider. ix, xi, xix, 9, 14, 15, 19, 26, 32–35, 37, 38, 40–47, 49, 56, 58, 59, 69, 82

PII Personal Identifiable Information. 34

REST Representational State Transfer. 6, 7, 18, 19, 21, 23, 24, 27, 29, 47, 48, 50, 51, 56–61, 64, 67–69, 75, 77, 82, 83

RP Relying Party. 7, 14, 15, 20, 34, 35, 37, 38, 40–43, 45, 47, 49, 56–59, 71, 82

RPT Requesting Party Token. xiv, xix, xx, 7, 23, 47, 49, 53, 56, 57, 59–64, 67–69, 71–75, 77, 78, 82, 83

SAML Security Assertion Markup Language. xix, 16, 17, 19–21, 29, 30

SDN Software Defined Networking. ix, xi, xiii, xix, 3, 6–9, 11, 18, 19, 21, 23–33, 35, 37–39, 63, 82

SEAF Security Anchor Function. 18

SHA Secure Hash Algorithm. 33, 34

SNMP Simple Network Management Protocol. 26, 28, 29

SP Service Provider. 16, 19

SSO Single Sign-On. xix, 14, 16, 17, 19, 21

TCI Trust Context Information. xx, 37, 42–47, 51–53, 63, 67, 69, 77, 80, 82, 83

TLS Transport Layer Security. 6, 9, 19, 24, 28

UA User-Agent. 7–9, 12, 33, 34

UDM Unified Data Management. 18

UE User equipment. 18

URN Uniform Resource Name. 8

UUID Universally Unique Identifier. 8, 33, 34, 58, 78

VM Virtual Machines. 34

This page is intentionally left blank.

List of Figures

2.1 Abstract OAuth 2.0 Flow 13
2.2 Authorization Code Grant 14
2.3 OpenID Connect Additional Layer 14
2.4 OpenID Connect Authorization Code Flow 15
2.5 Security Assertion Markup Language (SAML) Single Sign-On (SSO) 17
2.6 5G-AKA Authentication Flow 18
2.7 Software Defined Networking (SDN) Architecture 23
2.8 OpenDaylight Neon Architecture 25

3.1 Shiro-Based Authorization 36
3.2 MDSAL-Based Dynamic Authorization 36
3.3 SDN Architecture With OpenID Connect (OIDC) Integration 38

4.1 Evaluation Scenario 42
4.2 Evaluation Scenario Flow 43
4.3 Token Verification Time On TeaStore 45
4.4 Token Issue Time On OpenID Provider (OP) Per Policy 45
4.5 Context Verification Time On OP Per Policy 46
4.6 Overall E2E Time Per User 46

5.1 Roles Registration In OpenDaylight 49
5.2 Roles In Keycloak 50
5.3 *"admin-role"* In Keycloak 51
5.4 Scopes Mappings Of *"admin_cli"* Client In Keycloak 51
5.5 Resources Of The *"controller"* Client In Keycloak 52
5.6 Authorization Scopes Of The *"controller"* Client In Keycloak 52
5.7 Policies Of The *"controller"* Client In Keycloak 53
5.8 Permissions Of The *"controller"* Client In Keycloak 53
5.9 Scope Permission *"ScopeDomains"* Of The *"controller"* Client In Keycloak . 54
5.10 Resource Permission *"Controlador"* Of The *"controller"* Client In Keycloak 54
5.11 Payload Data Of Requesting Party Token (RPT) 56
5.12 Simplified Architecture OpenDaylight Controller 58
5.13 Simplified Modified Architecture OpenDaylight Controller 59
5.14 Authentication Steps 60
5.15 Verification Of RPT Token Steps 61
5.16 Authorization Data In RPT Token 61
5.17 Renewal Of RPT Token Steps 62

6.1 Renewal Timeline In A 2 Min Session 65
6.2 Get Access Token 71
6.3 Get RPT 72
6.4 Put User Atributes 72

6.5 Verify RPT 73
6.6 Size Of RPT 73
6.7 RAM Used (5 Group; 25 Group, 50 Group) 74
6.8 Get New RPT 74
6.9 RAM Used (5 Group; 25 Group, 50 Group) 75
6.10 Rest Operations With Authentication Time In OpenDaylight Standard And OpenDaylight With OIDC 76
6.11 Rest Operations With Authentication In Standard OpenDaylight And Only Rest Operations In Modified OpenDaylight 76
6.12 [AT-01] - Attack Tree In Standard OpenDaylight 78
6.13 [AT-02] - Attack Tree In OIDC OpenDaylight 79
6.14 [AT-03] - Attack Tree In OIDC OpenDaylight With Support For Trust Context Information (TCI) 80

This page is intentionally left blank.

List of Tables

2.1 Related Work Overview 21
2.2 Comparison AAA Support Between controllers 27

3.1 Research Inclusion Criteria 31
3.2 Trust Context Information Fields 33

4.1 Configuration Parameters 44
4.2 TCI Evaluation Metrics 44

5.1 Role-Endpoint Authorization Mapping 48
5.2 Keycloak Functional OIDC Clients 50
5.3 Authorization Configurations Summary 55

6.1 Scenarios Combination For Authentication And Request 64
6.2 Requests For Authentication Test Set 65
6.3 Requests For Authentication Test Set (Continue) 66
6.4 Type Of User Instances 67
6.5 Metrics For Authentication And Request 67
6.6 Scenarios Combination For Renewal 67
6.7 Requests For Renewal Test Set 68
6.8 Metrics For Renewal 68
6.9 Requests On Standard And Modified OpenDaylight Controller 68
6.10 Metrics For Standard And OIDC Opendaylight Authentication 68
6.11 Scenarios Combination For Access Denied Test Set 69
6.12 Requests For Access Denied Test Set 70
6.13 Attack Tree Comparison - OpenDaylight Security Integrations 79

This page is intentionally left blank.

List of Publications and Tutorials

The work in this dissertation regarding context information in OIDC and OIDC authentication in a SDN controller is partly based on the work presented in the following publication and submission:

- **Publication:** Carolina Gonçalves, Bruno Sousa, Nuno Antunes, "BIANFE: Object identification and authentication in federated scenarios", CCNC 2022, poster, January 10, 2022 (appendix A).
  - **abstract:** *Federated Identity Management enables convenient mechanisms to authenticate users and to authorize services, applications to specific users' resources. SAML and OpenID Connect that relies on OAuth 2.0 are commonly employed to enable Single-Sign-On features. Despite their wide usage in several domains (enterprise, web applications) they only aim to identify entities like persons and do not consider the different trust levels that a person can have with its devices, or even with the services provided by organisations participating or not in federated scenarios. BIANFE stands as a proposal for object identification and authentication in federated and non federated scenarios, considering the trust relations between end- users and the applications/services running in its devices. As work in progress, BIANFE tackles primarily the identification issue for objects, considering interoperability and privacy issues.*
- **Rejected but being reformulated for a new target:** Carolina Gonçalves, Bruno Sousa, Nuno Antunes, "TCI: Context Information for Trust in Federated Environments", Trust, Security and Privacy of 6G, IEEE Network Special Issue. **New target:** 14th IFIP Wireless and Mobile Networking Conference.
  - **abstract:** *Current and future networks must tackle identity management of users in federated scenarios, where the information of users is managed by a trusted entity. This model is widely employed nowadays in our daily lives, where we authenticate in third party services using account information in google or others that provide authorization solutions. Other solutions like OpenID Connect relying on OAuth 2.0 are commonly employed to enable Single-Sign-On features. Despite their wide usage in several domains (enterprise, web applications) they only aim to identify entities like persons and do not consider the different trust levels that a person can have with its devices and the context in which interactions occur. TCI is as a proposal to convey context information reflecting the trust relations between end-users, the applications/services running in its devices, and with particular contexts where the access to sensitive resources needs to be authorized. The results demonstrate TCI as a feasible solution to convey trust with minimal impact, in compliance with OpenID Connect.*
- **Tutorial:** Carolina Gonçalves, Bruno Sousa, Nuno Antunes, "Trusted Federated Identity Management in services and SDN", 27th IEEE Symposium on Computers and Communications .
  - **abstract:** *Federated Identity Management (FIM) is a topic that has attracted the research community and enterprises to build different solutions, suited to specific needs. A few examples include: the Security Assertion Markup Language*

(SAML), the Open Authentication (OAuth 2.0), and OpenID Connect (OIDC) as a solution to support authentication and identification of users.Identity management solutions include mechanisms and architectures to exchange identity information between organisations that are federated for authentication purposes. This has the advantage inherent to the Single Sign On (SSO) process, where an user does not need to replicate login information over multiple systems.The evolution of 5G towards 6G will enable multiple access contexts, associated with the support of heterogeneous networks, with the support edge and cloud computing models. In such context, current FIM solutions mainly focus identity management and do not consider the possible context environments where services require user information for authentication and authorization purposes.

Chapter 1

Introduction

The advancement of technologies, such as Software Defined Networking (SDN), has made network administration easier. The SDN technology offers more flexibility and allows the administrators to control networks, modify configuration settings, among others from a centralized location. This technology has also been used in cloud computing and telecom operators facilitating the planning and operationality of communication networks.

The SDN architecture with a centralized controller has constraints on the flow tables of various devices, which tend to be vulnerable to distributed denial of service attacks (DDoS) [1, 2] and other attacks. SDN has become an easy target since it centralizes all respective network information. There are currently some security features that can be employed to try to prevent these problems such as access control lists, "honeypot" for SDN applications and the use of Transport Layer Security (TLS) [3].

SDN controllers can have several applications with different features, such as metrics collection, and dynamic flow management, among others [4]. These applications are installed using the management Application Programming Interface (API) provided by the controller (Web interfaces, Representational State Transfer (REST)-based interfaces, Command-line Interface (CLI)). These interfaces rely on basic authentication schemes, with username and password, which in cases where the credentials are stolen either by social engineering, through spoofing, or another exploit, and these attacks are not detected, the attacker will have unlimited access to the controller, its resources and will have control over the respective network. Therefore, **we need a more robust authentication mechanism in SDN controllers to prevent such threats.**

OpenID Connect (OIDC) [5], a federated authentication based on the OAuth 2.0 protocol [6], is an example of a protocol that can be a better-suited solution to solve the stated problem and increase the security of SDN controllers since it can both authenticate and authorize a user. Both OIDC and OAuth 2.0 protocol use access tokens, which are tokens that allow the user to access the restricted resources in the controller for a limited time, limiting the attackers' access even if a token is stolen. These protocols are widespread in the Internet, for authentication and authorization in websites via *Google* or *Facebook* accounts.

Some SDN controllers started to incorporate Authorization Authentication and Accounting (AAA) services to have a better user management, including an integration with protocols such as OAuth 2.0. The integration with this protocol can be found on SDN controllers including ONAP [7] and OpenDaylight [8].

The major goal of this work is the design and evaluation of an authentica-

tion and accounting framework for SDN networks with the integration of the OIDC and OAuth 2.0 protocols. The flexibility of these protocols also presents the **opportunity to add more information regarding the authenticated devices of an end-user**. This opportunity allows the end-user to have **different levels of trust regarding its devices**, the User-Agent (UA) (e.g *Google Chrome*), service, application and network used. For example, a user can have full trust in an environment with a private network, while using their personal computer where the UA is *Google Chrome* whereas in an environment with a public network the same level of trust cannot be matched.

Current approaches frequently have drawbacks whether using basic authentication with a username, password combo, or using OAuth 2.0 with the use of basic tokens (like standard OpenDaylight). These basic tokens are a Base64-encoded username and password, therefore, do not provide a strong security level. The use of bearer tokens, meaningless non-signed strings which do not have encoded credentials, or JSON Web Token (JWT), signed and encrypted tokens with a JSON payload that includes information regarding the authentication process and some user information, would be an ideal solution to raise the authentication robustness of the SDN controllers and consequently improve the security of the network.

In this work, we start by understanding the impact of adding the context information regarding the authenticated devices of an end-user in a OIDC flow, primarily in a web context, since most OIDC integrations tend to be in that context. So, we propose a methodology, Section 3.3.1, in which we indicate the fields that represent the context information, as well as an algorithm to evaluate those fields and enforce policies accordingly. The preliminary results of this methodology revealed that the flow of authentication with OIDC needs to be adjusted to support the context information and the additional fields **did not add a high cost in the REST operations and authentication flow**.

To start developing the framework proposed, first is presented an architecture for the framework, highlighting the main components, OpenDaylight controller, AAA filter, Relying Party (RP), and Keycloak, as well as describing the interactions between them. In Chapter 5, we introduce the concepts of roles and grants on the controller, and how we take advantage of those concepts to have a more refined authorization to the different REST endpoints of the controller. We also describe the configurations on Keycloak and the modifications done on the AAA filter of the OpenDaylight controller, together with the creation of the component RP in the OpenDaylight controller.

For Keycloak we detail the OIDC clients created as well the configuration for each one, as well as introduce the concept of an Requesting Party Token (RPT), Section 5.2.1. For RP we explain how it uses the Keycloak REST API, to create a bridge of communication between the controller and Keycloak in Section 5.2.2. Section 5.2.3 describes the work of the AAA filter and how it manages the received requests. A suggestion of a new architecture for OpenDaylight is also proposed where the AAA filter is common integration for the different modules of OpenDaylight, even though its full integration is out of the scope of this work. At last, we explain the flow of an OIDC authentication with context information, how the authorization bearer token is verified together with the context information, and to proceed when a bearer token needs to be renewed.

To evaluate the framework proposed, we prepared an experimental procedure to test different parameters related to authentication, including the verification of a token sent by an authenticated user, parameters related to the token renewal, compare the behavior of the same user, "admin" in both an OpenDaylight standard, and the OpenDaylight of this framework which was modified, and at last, analyze the security and functionality of the roles added in OpenDaylight and the use of context information to make sensible opera-

tions to the controller. This experimental procedure included a different number of users, different types of users, and different context information.

The results obtained state no significant differences for the metrics collected regarding the authentication or renewal procedures, where in most cases even with a different number of users, or user's type, or context information the values obtained were very similar (complete results in Section 6.2). The test set related to security and functionality of roles and context information showed that the roles added prevented unauthorized requests and the use of context information prevented sensible operations to be realized under scenarios with low and average trust. However, when comparing the timing metrics obtained from the controller of this framework with the standard controller within the same user, the timing of authentication is almost 8 times higher, as well as operations in the controller that will also update Keycloak database (e.g creation of a user), are almost 3 times higher.

Even though there is a cost to increase the level of security in the OpenDaylight controller, all the times obtained were in milliseconds (ms), and none of the values presented went above a second, which indicates that in a user experience while interacting with the controller, **this cost will go almost unnoticed while experiencing twice the security**, benefiting from the integration of OIDC and the use of context information. By following a methodology of using attack trees to compare different authentication mechanisms while analyzing the threat range and complexity of the tree, we showed the security gain, since the attack trees presented an increase in the complexity of the attacks with each security element introduced in OpenDaylight.

1.1 Objectives

The main objective of this work is to design and evaluate a framework, which incorporates authentication and accounting mechanisms in SDN controllers. This objective aims to identify and authenticate end-users who want to access resources in a SDN controller.

To accomplish this goal a security feature such as OIDC was integrated in an SDN controller - OpenDaylight, with an extension to new parameters to regulate the levels of authorization based on context information of end-users regarding its devices.

1.2 Contributions

The contributions presented are the ones achieved during the development of this work.

- The current Federated Identity Management (FIM) solutions are primarily concerned with identity management and do not take into account the context environments in which services may require user information for authentication and authorization.
  - The proposal of a solution for object identification and authentication in federated scenarios, taking into account the trust relationships between end-users and applications/services operating on the devices. These 'objects' can be a physical device or an application, with associated environments, on which UA are executed. The Uniform Resource Name (URN) and Universally Unique Identifier (UUID) standards would be used to identify such 'objects'.
  - The proposal of a methodology that takes into account the trust context information of a trustor, user, and trustee services, networks, devices, and applica-

tions in federated environments to address constraints in FIM solutions based on OIDC, since OIDC simply seeks to identify entities and ignores the many levels of trust that a person might have with their devices as well as the environment in which interactions take place. For example, a user may have greater trust in a home assistance device than in an *IoT* device that measures temperature on their home network.

– The proposal of a solution by adding a set of fields to convey context information expressing a user's trust in a device, or in a UA accessing services through networks. It considers the approach of adding a new claim in OIDC. Aside from establishing basic claims, the OIDC specification also allows for the provision of extra claims in a standard manner. The verification of this new claim will result in distinct policies at the OpenID Provider (OP), providing differentiated access to the resources, in this example, registered user personal information.

- The implementation of OIDC support with an extension for context information in TeaStore [9] (a basic web store). The experimental procedure in Chapter 4 portrays the implementation and evaluation of this contribution.
- The implementation of Hypertext Transfer Protocol Secure (HTTPS) in TeaStore. For the experimental procedure explained in Chapter 4, all entities involved used HTTPS to communicate with each other. As TeaStore was not able to support HTTPS and share a TLS certificate for the 6 services it includes, there was close work done with one of the authors of TeaStore, in order to make it support HTTPS. Now, TeaStore can support HTTPS communications, which work can be found in the *Github* main repository of TeaStore [10].
- Implementation of the framework proposed, OpenDaylight controller with support for OIDC authentication as well an extension for context information, while using Keycloak as the OP and to have better user management, Chapter 5.

1.3 Structure

The remaining document is organized as follows:

- Chapter 2 - Provides an overview of the key topics covered by this work, introducing some concepts such as security features, SDN controllers, and protocols. It also reviews the related work regarding security features implementations in Web or SDN.
- Chapter 3 - Describes the work's research objectives and introduces the approach taken throughout the dissertation work, as well as the methodology used.
- Chapter 4 - Presents the experimental work accomplished in the first semester.
- Chapter 5 - Describes the work conducted in the second semester, and the implementation of the framework proposed.
- Chapter 6 - Presents the results obtained from the experimental work on the framework, as well as an analyze
- Chapter 7 - Provides the main conclusions, a summary of the results, and the future work that can be followed.

This page is intentionally left blank.

Chapter 2

Background and Related Work

This chapter provides an overview of security features, as well as the related concepts that are relevant to understanding the topics covered in this thesis. Their functionalities and characteristics are detailed in order to understand the flow of data while using such security features, the actors involved, and the objects traded for each request and response.

Five Software Defined Networking (SDN) controllers are introduced, as well as an overview of their support for Authorization Authentication and Accounting (AAA) services, additional features, or relevant plugins denoting this controller, and in some cases their architecture.

Different SDN protocols are also presented to comprehend the communications and operations occurring with and within an SDN controller and respective switches/devices in the network. The benefits of such protocols are also detailed.

At last, even though there is limited research from our understanding regarding the implementations of the security features presented in SDN controllers, there is somewhat research regarding the security features in web implementations. Therefore, in the following sections, we also discuss related work to both of these two types of implementations.

2.1 Security Features

Overall, security features have the objective to increase the security of a process, operation, application, or communication. The security features presented in this section aim to increase security regarding the authentication and authorization process of individuals. These features also take extra measures to ensure that the authenticated individuals accessing restricted resources are legitimate and not some attacker bypassing security processes. Some of these features are more tailored to web implementations and others to network implementations.

2.1.1 OAuth 2.0

OAuth 2.0 is an authorization protocol for access delegation that is defined in RFC 6749 [6]. This security standard is only designed for authorization where a user permits applications to access its data. The user can also grant access to other applications on his behalf, without the need to use passwords. Authorization or delegated authorization are terms used to describe the steps involved in the granting permission process. OAuth 2.0 also

defines measures to revoke the permissions that were given.

In OAuth 2.0 several entities interact with each other [11]. The *Resource Owner*, often known as the end-user, is the person who is in charge of the resources. Following, there is the *Client*, the consumer of the resources, which is an application that looks for the end-user approval to access the data on their behalf. The Application Programming Interface (API) that the *Client* wants to access on behalf of the end-user is called a *Resource Server*. It controls access to resources and who has access to them based on a set of rules. The *User-Agent (UA)*, which is typically a browser, that interacts with the client to provide the required information for authorization requests and is used by the *Resource Owner* to communicate with the Authorization Server (AS). Finally, the AS is an application in which the *Resource Owner* already has an account. The AS should be able to verify a *Client*'s identification and, if necessary, authenticate the end-user.

To fully grasp the concept of OAuth2.0, some concepts need to be explained:

- **Scope** - A scope defines which data the *Client* application needs to access. If a *Client* needs access to multiple resources, there will be multiple scopes defined.
- **Consent** - When the AS receives the scopes in the request of the *Client*, it will verify with the *Resource Owner* if they want to give their consent to the *Client* to access the data in those scopes.
- **Redirect URI** - Or callback URL, is the URL that the authorization server will redirect the end-user to, after giving their consent to the *Client*.
- **Response type** - Response type is the type of response the *Client* is expecting to receive. In the case of the Authorization code grant, the response type is code.
- **Authorization code** - An authorization code is a temporary code that the *Client* gives the AS in exchange for an Access Token.
- **Access Token** - An Access Token is a ticket that gives permission to the *Client* to communicate with the *Resource Server* and retrieve data on behalf of the end-user. The most common access token type used in OAuth 2.0 is the bearer.
- **Refresh Token** - A Refresh token is a token used for renewing an access token after its expiration.

Figure 2.1 [12] represents the abstract OAuth 2.0 flow, with the entities mentioned and the interactions between them. Using the example of a web application, the *Client*, that offers log-in via *Facebook*, it first needs to make an authorization request to the resource owner, the end-user. The *Client* then receives an authorization grant, which is a credential associated with the end-user authorization. Following this first request, the *Client* makes a token request to the *Facebook* AS, while presenting the authorization grant received. The *Facebook* AS when it receives this request, authenticates the *Client* and verifies the authorization grant, responding with an access token if the authorization grant is valid. With this token, the *Client* makes a request to the *Facebook Resource Server*, which includes the access token and the resources it wants to access. The *Facebook Resource Server* validates the token and upon successful verification, responds with the requests fields. The *Client*, with the information received, can then authenticate the end-user into their application.

When developers register a *Client* in the AS, the server generates two components: a client ID and a client secret, which are used in all subsequent OAuth transactions. The client

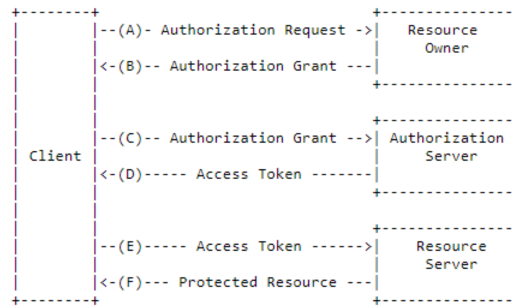


Figure 2.1: Abstract OAuth 2.0 Flow

secret is similar to a password that only the AS and *Client* know in order to communicate with each other. The client ID identifies the client in the AS.

Grant types are one of OAuth 2.0's most important features. A grant type specifies how a *Client* application obtains an access token from an OAuth 2.0 AS. This access token allows a client application to access a resource on behalf of the *Resource Owner* [13].

The main grant types defined in OAuth 2.0 are the following:

- **Authorization code grant** - In this grant, there are two requests directed to the AS. The first request which response is an Authorization Code, and the second request exchanges the authorization code for an access token
- **Implicit grant** - In this grant, there is no intermediate step of exchanging an authorization code for an access token. This token is given in the first request. When the AS issues an access token, it does not authenticate the *Client*.
- **Resource owner credentials grant** - In this grant, the *Resource Owner*'s credentials, username, and password are exchanged in a single request for an access token.
- ***Client* credentials grant** - In some cases, where the *Client* is also the *Resource Owner*, the *Client*'s credentials (client ID and a client secret) can be used as an authorization grant, and exchanged for an access token.
- **Refresh Token grant** - This grant is used to retrieve new access tokens when the current access token expires

An OAuth 2.0 grant type is represented by numerous requests and responses in order to receive an access token from the authorization server to access the protected resources. For example, as mentioned, the Authorization Code grant is divided into two parts [14]. Each part requires an endpoint in the AS to be dealt with it, as shown in Figure 2.2 [14]. The first request, the authorization request, will be directed to the authorization endpoint in the AS. In this request, along with the client id, and client secret, other parameters are specified such as the scope that the token is expected to have. A token can have access to different data items within the *Resource Owner*, the end-user, hence the possibility of the definition of multiple scopes. However, the AS does not have to issue a token with all requested scopes if it is not necessary or if the *Resource Owner* does not give consent to the presented scopes. The second request, the access token request, is handled by the token endpoint in AS, where there the *Client* exchanges an authorization code for an access token, with the scopes defined associated with it. At last, with this token, the *Client* can access the protected resources on behalf of the end-user.

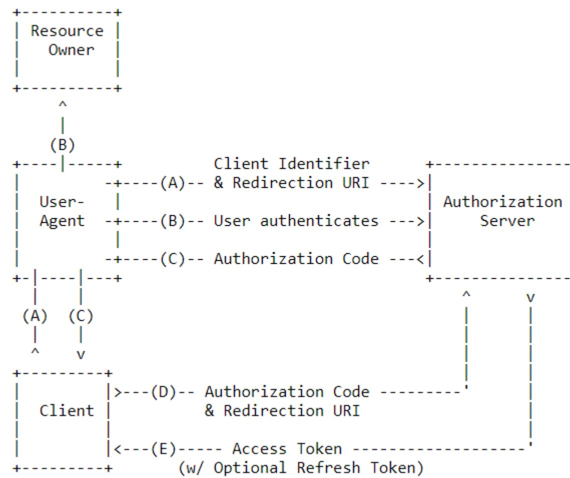


Figure 2.2: Authorization Code Grant

2.1.2 OpenID Connect

OpenID Connect (OIDC) is an additional security layer on top of OAuth 2.0, Figure 2.3 [15], and defines how OAuth 2.0 can be extended to identify a user. OIDC adds both profile and login information about the person who is authenticated. OIDC facilitates Single Sign-On (SSO) (single login used across numerous applications/services), identity federation, attribute sharing, single logout, and other use cases [13].

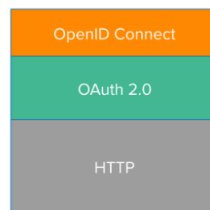


Figure 2.3: OpenID Connect Additional Layer

In OIDC, there is the redefinition of some entities. The OpenID Provider (OP), which corresponds to the OAuth 2.0 AS, is responsible for the authentication, obtaining the consent and authorization of the end-user. The *Relying Party (RP)*, which corresponds to the OAuth 2.0 *Client*, is in charge of establishing a connection with the end-user to handle the information required for the authentication.

The OIDC scope is a new feature introduced by OIDC that allows authentication as an extension of the OAuth2.0 protocol. OIDC extends OAuth with the ID Token extension, which is a security token, a JSON Web Token (JWT), with multiple claims about the authentication of an end-user [5]. A claim corresponds to some information regarding an entity that has been stated. Even though JWT is not comprehensive just by looking, the *RP* can extract information from it, such as the ID, name, the time of issuing the access token, the ID Token expiration, and verify if the JWT has been tampered with.

The mandatory fields included in the claims of an ID Token are the following [5]:

- **iss** - Is a URL that includes the token issuer's scheme, host, and port number.
- **sub** - Is a unique subject identifier with local context for the end-user.

- **aud** - It corresponds to the OAuth client id, identity to which the ID token is intended for.
- **exp** - Expiration time.
- **iat** - Time on which the token was issued.

Authentication flows are the methods designed for a *RP* to obtain an access token and an ID token. An authentication flow is more than a grant type even though it uses grant types. In OIDC there are three main flows Authorization code flow, Implicit flow, and Hybrid flow.

The authorization code flow in OIDC, Figure 2.4 [16], is similar to OAuth 2.0 Authorization code grant. The main differences are that scope of OIDC is specified in the first request, and the *RP* receives both an Access Token and an ID Token in the final exchange. The *RP* must communicate with the OP's token endpoint to obtain an ID Token and an access token. There is also a UserInfo Endpoint, in the OP, where a request made with an access token will result in the returns of claims about the end-user [17].

At the beginning of the authorization code flow, after verifying the authentication request from the *RP*, the OP always verifies whether the user is authenticated, if it has a valid login session under the OP's domain or if it has already logged into the OP using the same web browser [18]. If the user does not have a valid login session, the OP will prompt them to authenticate, as well as obtain their permission to share the required claims with the *RP*.

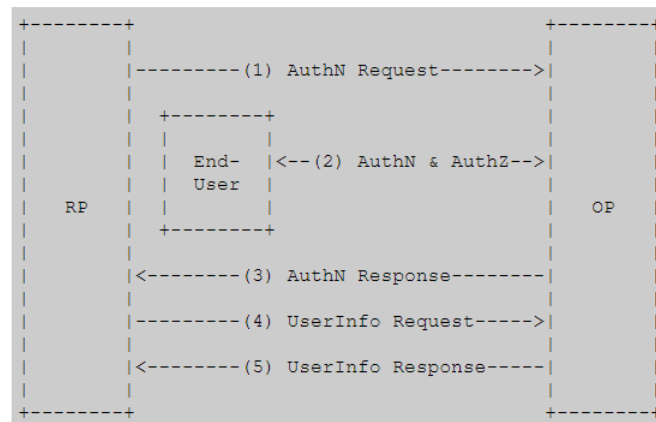


Figure 2.4: OpenID Connect Authorization Code Flow

The authorization code flow has specific characteristics:

- The redirect URI in the authentication request has to match the one that is already registered in the OP.
- The value of the state parameter, a simple string added to the authentication request, must be returned to the *RP* by the OP in the authentication response.
- The nonce field contains a unique value that the *RP* adds to the OIDC authentication request.
- There is only one possible value of response in the authentication code flow, which is the code.

Nowadays, the main usage of OIDC is for the sign-in process in web applications. Websites such as *SAPO, Yahoo, WordPress, Notion, Trello, Medium, Zoom (Google or Facebook)* and many more [19], support sign-in via identity providers such as *Apple, Facebook, Google, Linkedin, Microsoft*, which makes the process of sign-up into an application much easy and fast, as well lessen the need to create individuals accounts for each web application.

2.1.3 SAML2.0

Security Assertion Markup Language (SAML) is an XML-based framework for sharing security information between business groups, with the use of a SAML assertion payload. The SAML 2.0 has been embraced in many enterprise environments since it allows their applications to authenticate and keep the user's digital identity in a centralized identity provider [13]. Cross-domain SSO and identity federation have become the two major elements of SAML 2.0.

In SAML there are three key actors in any interaction. The first, Identity Provider (IdP) authenticates and authorizes users. It is a trusted organization and issues SAML assertions tokens about an authenticated subject. A subject is an entity whose security information will be exchanged, which is usually the application's user. When a subject is authenticated, the results of the subject authentication event are returned in an XML message known as an authentication response. The authentication assertion payload in this response contains claims about the authentication and the authenticated user.

The second actor is a Service Provider (SP). It delegates subject authentication to a trusted IdP. Additionally, acts on information encoded in assertion tokens about an authenticated subject to decide whether this subject is or not allowed to access a resource.

The User is the third and last actor, which can be a program that wants to access a resource. The user is usually the first to initiate the protocol by communicating with the SP.

An assertion is an XML-based statement that conveys information regarding a subject's security. A SAML assertion is a claim or statement made by the IdP and trusted by the SP. There are three types of assertions:

- An authentication assertion that verifies the user's identity.
- An attribute assertion that carries user-specific information.
- An authorization assertion that specifies what the user is authorized to do.

Cross-domain web SSO is one of the most common SAML scenarios. The Figure 2.5 [13] shows a SAML SSO where a subject starts by wanting to use an application that serves as a SAML provider. However, before this interaction can begin, the SP and the IdP must exchange information to configure and set up a trust relationship. Additional information can be requested after the initial transaction (these two steps mentioned are not presented in Figure 2.5). As a result of the initial interaction, a SAML request is generated, as presented in step 2 of the figure, and user authentication is delegated to an IdP. This IdP interacts with a user to verify credentials and authenticate them. If successful, the application receives a SAML assertion. The subject logs in once the program validates the SAML assertion in the response.

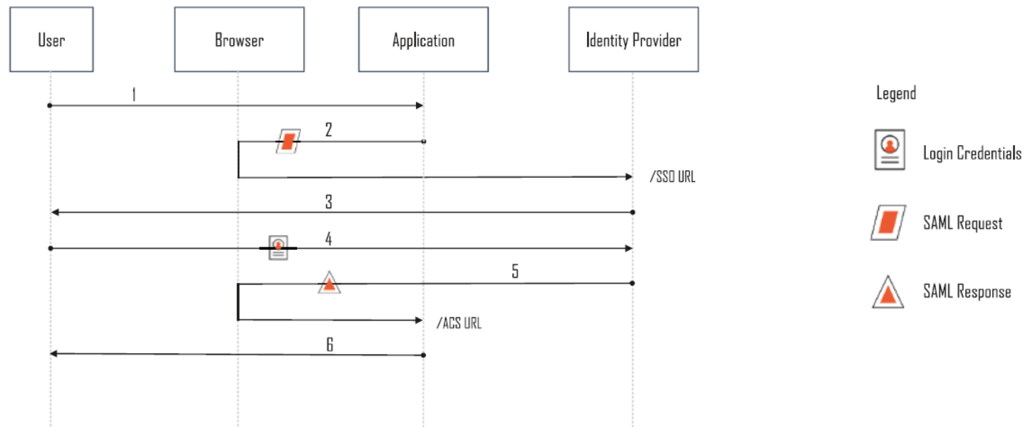


Figure 2.5: SAML SSO

Identity federation is also a method offered by SAML, as mentioned, where an IdP and application use a commonly shared identifier for a user. This can be used internally or across systems.

These days, web applications such as *Educast, Zoom (SSO), IEEE Xplore*, support SAML 2.0 and consequently SSO, where it is possible to have a federated authentication with the credentials of an institution, including the University of Coimbra.

2.1.4 Authorization Authentication and Accounting

AAA stands for authentication, authorization, and accounting and is a framework to provide an additional layer of security. This framework is extremely relevant in network management and security because we want to prevent unauthorized users to access network resources and perform actions that require authorization, as well as preserving sensitive information when necessary. AAA also helps with non-repudiation, since an individual cannot deny their actions if authentication, authorization, and accounting are in use.

Therefore, AAA controls who is allowed to use network resources, and what they are authorized to do, and also records the actions done by one accessing the network.

When a user wants to access network resources, the first step is to perform authentication to identify the individual. In most cases, authentication is done by providing a valid and unique username and password. In other cases, authentication can be done through a 3rd party, an external AS. The user's credentials provided are compared with the stored user's credentials. If the credentials do not match, it is denied access to the network. However, if credentials match, the user is authenticated and gains access to the network.

Following a user authentication, it is necessary to verify its identity and decide which resources this user should have access to or which operations is the user authorized to perform. Multiple authenticated users may have different levels of authority in the system. Authorization helps to decide what Access Control List must be applied, or which role the user belongs to.

The final step in the AAA framework is accounting. Accounting or auditing keeps track of a user's activity and what resources the user consumes during its access, the time spent in the network as well the services accessed. This logging of information is used for authorization control, resource utilization, and analysis of the system.

Since SDN is one of the key technologies of 5G, authentication is also critical to the security of cellular networks since it allows users and the network to establish mutual authentication. In the 5G Architecture designed by 3GPP [20], there is an Authentication Server Function (AUSF) that eases these security processes. In the home network, the AUSF handles authentication with a User equipment (UE). It decides on UE authentication, but it uses the help of a backend service to handle the additional computation required for the authentication data. Unified Data Management (UDM) is the backend entity that picks an authentication method based on the identifiers received and policy settings, as well as computing the authentication data needed for AUSF. Finally, there are the Security Anchor Function (SEAF), which resides in a serving network and acts as a "middleman" between a UE and its home network throughout the authentication process [21].

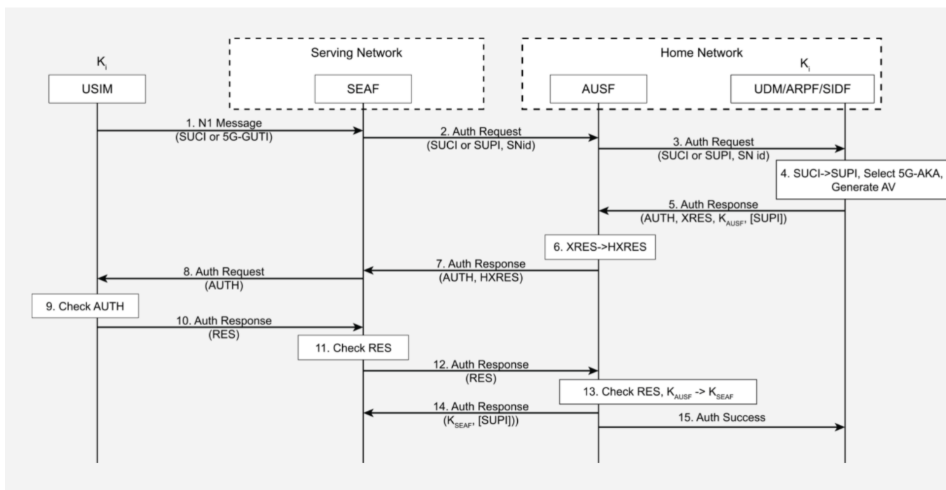


Figure 2.6: 5G-AKA Authentication Flow

For example, in the 5G-AKA authentication flow, Figure 2.6 [21], after receiving any signaling message from the UE, SEAF may begin the authentication procedure by sending an authentication request to the AUSF. The AUSF checks to see if the providing network requesting the authentication service is authorized. The AUSF sends an authentication request to UDM if it is successful. The authentication method is selected, in this case, 5G-AKA, after receiving this request and the data in the request is decrypted [21]. UDM initiates 5G-AKA by delivering an authentication vector including an AUTH token to the AUSF. The AUSF generates a hash of the expected response token and transmits the authentication response, as well as the AUTH token and this computed hash, to the SEAF. The SEAF saves the hash and delivers the AUTH token to the UE as part of an authentication request. Using the secret key it shares with the home network, the UE validates the AUTH token. The UE considers the network to be authenticated if validation passes. The UE completes the authentication process by calculating and delivering a token to the SEAF, which the SEAF validates. The SEAF then sends the token to the AUSF for validation if the token is successful [20]. If the UE's token is valid, the AUSF generates a key and delivers it to the SEAF, as well as notifying UDM of the authentication results.

2.2 Related Work

In 2015, Oktian et al. [22] proposed a more secure Representational State Transfer (REST) Northbound Interface for SDN controllers, using token-based authentication and authorization for applications and users, based on OAuth2.0 protocol (more about SDN in the next

section, 2.4). They discuss the current implementation of OpenDaylight, where a user exchanges username and password information for an access token, and other implementations including Floodlight, Open Network Operating System (ONOS), Ryu, or POX which offered no REST Northbound Interface security. However, the approach of OpenDaylight can only authenticate the user but not the application. So, their proposal uses delegated authentication flow, in which the application contributes to the authentication process, authenticating both the user and the application. Their proposal also provides *ID-Based Cryptography* mechanism to replace Transport Layer Security (TLS) and JWT to give additional security in REST Northbound Interface Messages.

Following this work in 2017, Oktian et al. [23] presented a REST API security framework for SDN controller based on OAuth 2.0, called OAuthkeeper. Their prototype was implemented on top of ONOS SDN controller and used Bearer Token, which was stored in the memory of SDN controllers, and offered support for the four grant types provided in OAuth 2.0 protocol. Additional OAuth 2.0 parameters, such as domain, scope, switch, and controller, were also supplied by the authors in order to provide more tight access control. There was an adaptation to the filter module present in ONOS, which was offered to all REST API endpoints. This filter includes three types of authentication filters, one for basic authentication, one for form authentication, where it detects if there is a session present in an Hypertext Transfer Protocol (HTTP) request, and if not OAuthkeeper will redirect the request to a login page, and at last a token authentication filter, to retrieve an access token from the HTTP request and verify them. They've also carried out practical tests with the goal of determining how many REST API requests per second ONOS can manage with their framework incorporated, where they used used *Mininet* as their network simulation tool to simulate SDN switches and hosts, and Locust to generate a flood of users to simulate REST API requests from SDN applications to ONOS. They also tested scenarios with ONOS Basic authentication and ONOS without any REST API security. Their framework presented a good performance, with minimal overheads, where their work produces a 15% overhead from the unsecured ONOS to a 3-4% overhead compared to the default ONOS Basic authentication.

Bonelo [24] work provides a method of integrating dynamically deployed services into an SSO provider in a federated network of users. CYCLONE-PAM is an authentication system that allows users to log in to an SSH server using their institution's SSO credentials. Their solution is built on extending the PAM process to enable the use of OIDC as an authentication source. With this approach, their main goal was to provide a user experience comparable to the one provided by the use of RSA keys, only in this case while using web browsers and SSO technologies. Only JWT client authentication is supported in their implementation.

In 2020, Holtmann [25] contributes with a thorough examination of OIDC implementations and security elements of SSO. Their experimental architecture was mostly Docker-based, with Burp Suite serving as an intercepting proxy for the local test environment. For their custom OIDC implementations, the author also employed NodeJS webservers. Four OPs and five SP implementations were assessed for security throughout the evaluation process, using known and specific attacks against OIDC implementations. Two specification inaccuracies were discovered, both of which potentially have security ramifications. More typical concerns were found with CRLF injections in OP supplied data and Server-Side Request Forgery.

In 2017, Naik et al. [26] presented an evaluation of the SAML, OAuth 2.0, and OIDC standards in terms of security resilience and vulnerability, design, and functionality. The goal was to determine the right use of these standards to protect both the credentials

and digital identity of a user. It also provides an in-depth examination of their security vulnerabilities. Three separate attacks were carried out in the experimental phase: Denial-of-Service, Man-In-The-Middle, and Cross-Site Scripting. SAML shows to be vulnerable to Denial-of-Service attacks and appears to be vulnerable to the other two assaults as well. OIDC also had certain vulnerabilities, which were mostly caused by bad OAuth 2.0 implementations or logical flaws, or by the presence of a malicious OIDC identity provider.

In 2020, Basney et al. [27] used CILogon, an observatory to monitor authentication difficulties in a distributed identity federation, to investigate the underlying causes of authentication failures in the system where SAML IdPs and OIDC RP are serving academic research applications. The findings revealed that OIDC had almost double the failure rate of SAML, despite OIDC's claimed simplicity. To mitigate the errors found, it was also demonstrated that better error messaging may considerably reduce error rates by allowing problems to be detected and treated more quickly.

In 2017, Méndez et al. [28] presented OpenStack, an open-source cloud software solution that provides identity services (authentication and authorization) to other modules in OpenStack. With the use of ABFAB technologies (technology built on AAA, EAP, GSS-API, and SAML), OpenStack can be incorporated into a federation system with an AAA infrastructure. This OpenStack federation integration is accomplished by integrating the ABFAB Relying Party's capabilities into Keystone's Apache frontend. The authors also go through the authentication and authorization workflow, explaining how data is transmitted between the different entities. The AAA protocol is used to create the trust relationships between the RP and the IdP . At last, the results of the performance analysis suggest that the average authentication would be less than 2 seconds in production scenarios.

In 2020, Apte et al. [29] analyzed various protocols for Federated Identity Management (FIM), such as SAML, OIDC, and OAuth. These protocols are then evaluated and contrasted in terms of privacy and OpenStack implementation. During the evaluation, the authors determined that SAML shares identity attributes without the user's knowledge, and stated that user authentication stayed valid even when expressly removed in some circumstances, posing a greater risk for cross-domain deployment. On the other hand, OAuth and OIDC user identities were accessible to other parties after obtaining user consent, and they could be revoked at any time when the user's access to specified resources is removed.

Grassi et al. [30], authors of the National Institue of Standards and Technology (NIST) Digital Identity Guidelines, recommend the use of SAML or OIDC technologies to preserve the confidentiality of a subscriber's sensitive information (their digital identity), since these technologies use respectively assertions and claims, which can also be optionally digital signed.

Table 2.1 presents the related work compiled to provide an easier overview of the different research works.

2.3 Authentication and authorization tools

2.3.1 Authlib

Authlib is an open-source Python library to build OAuth and OpenID Connect clients and servers [31]. It offers generic but flexible implementations of RFCs, which include JWS, JWE, JWK, JWA, JWT, Oauth 2.0 OIDC and more.

Authlib provides support for many framework integrations including Flask, Django, Re-

Table 2.1: Related Work Overview

Authors	Summary
Oktian et al. [22]	• Secure REST Northbound Interface for SDN controllers • Token-based authentication and authorization for applications and users
Oktian et al. [23]	• REST API security framework for SDN controller based on OAuth 2.0 • Performance analysis
Bonelo [24]	• Authentication system to log in to an SSH server with institution's SSO credentials • Provide similar user experience as using RSA keys but with web browsers and SSO technologies
Holtmann [25]	• Analysis of OIDC implementation and security elements of SSO • Vulnerability assessment
Naik et al. [26]	• Evaluation of the SAML, OAuth 2.0, and OIDC regarding security strength and security vulnerability • Vulnerability assessment
Basney et al. [27]	• Identify authentication failures in system using SAML and OIDC • Monitorization of authentication difficulties
Apte et al. [29]	• Analysis of protocols for FIM, such as SAML, OIDC, and OAuth • Privacy assessment
Méndez et al. [28]	• integration of OpenStack in a federation system with an AAA infrastructure, using ABFAB technologies • Performance analysis
Grassi et al. [30]	• Recommend the use of SAML or OIDC to preserve digital identity confidentiality

quests, and more. Besides good documentation, Authlib already provides some examples of framework integrations with OAuth 2 and OIDC acting as a server and a client, which can be easily used and modified to support your project.

2.3.2 Keycloak

Keycloak is an Open Source Identity and Access Management which provides SSO, identity brokering (OIDC and SAML 2.0 Identity Providers), social login, User Federation (LDAP or Active Directory servers), an Admin and Account Management Console, support for standard Protocols (OIDC, Oauth 2, SAML) and clustering, and offers fine-grained authorization services [32].

Keycloak has an H2 database embedded however it offers support for Mariadb, Mssql, Mysql, Oracle, and Postgres. H2 is a very fast database engine and open source. It supports transactions as well as multi-version concurrency, allows full-text search, and

offers support to Java functions and stored procedures. H2 in an embedded mode offers faster and easier connections. At last, H2 has strong security features, having encrypted databased with Advanced Encryption Standard (AES). In addition, Keycloak has two types of cache. Local cache to store realm, client, role, and user metadata. The second type of cache stores offline tokens, user sessions, and login failures.

In Keycloak, a set of users, credentials, roles, groups, OIDC clients, and more are encapsulated in a Realm. Each Realm has its own set of users, and roles,.. and is only able to authenticate the users in their Realm. Roles in Keycloak are divided into three categories:

- **Realm Role** - Roles that belong to a specific realm. Accessible from any client and can be mapped to any user.
- **Client Role** - Role that belongs to a specific client. Only accessible from that client and only mapped to users of that client.
- **Composite Role** - Role which encapsulates more roles (realm or client roles).

As mentioned, Keycloak offers different access control mechanisms and fine-grained authorization policies for their resources [33]. Regarding a OIDC client, when the Authorization services are enabled, we are presented with the following concepts:

- **Resources** - Protected resources.
- **Authorization Scopes** - Actions performed on the resources.
- **Policies** - Conditions that must be fulfilled before granting access to a resource.
- P**ermissions** - Associate the resources with policies. These policies must be evaluated to decide if access should be granted.

A resource is a protected resource and it can simply be identified by its name. Authorization Scopes are also identified by a string, usually represented by a verb, get, view, delete, since it represents the actions that can be performed on a resource.

There are 9 types of policies, but the more relevant types for this work are Role Policy and JavaScript Policy. Role policy defines a user role, that can be later associated with a resource. Therefore, to gain access to that resource a user must have the role specified in the policy. The JavaScript Policy, allows the flexibility to coding any policy that must return **"$evaluation.grant();"** if the conditions are fulfilled. JavaScript policies are disabled by default in Keycloak, so is necessary to deploy a JAR to enable these scripts.

Permissions have two types, Resource Permission or Scope Permission. A Resource Permission is defined as the resources to protect using one or more policies. A Scope Permission, besides defining resources and policies, can also define scopes to associate with that resource and therefore control the actions that can be performed to that resource. For any type of Permission, we must choose a decision strategy. The decision strategy dictates how the policies associated with given permission are evaluated and how a final decision is obtained.

- **Affirmative** - at least one policy must be evaluated as positive for the final decision to be also positive.
- **Unanimous** - all policies must be evaluated as positive for the final decision to be also positive.

- **Consensus** - the number of policies evaluated as positive must be greater than the number of policies evaluated as negative. In case of a tie, the final decision will be negative.

Besides an access token, refresh token and id token generated by an OIDC authentication, Keycloak can also generate a Requesting Party Token (RPT) token (not on the specification of OIDC), which is an access token with all permissions granted by the OIDC client. These permissions (or authorization data) include the resources and scopes of that OIDC client that a user has permissions to access. To obtain an RPT, a request to the token endpoint must be made with the access token previously received in the authentication. This type of token results in fewer authorization requests since all the permissions needed for a user is already included in that token.

2.4 Software Defined Network Controllers

SDN represents a new approach for network management that abstracts network resources for applications and eliminates the need to repeat the same configurations in multiple devices [4]. Nowadays, with the increase of data centers, SDN's highly scalable and centralized network control architecture can offer significant benefits and cost reductions.

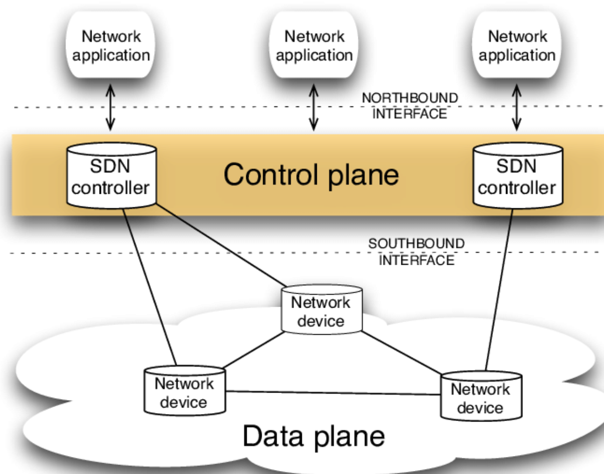


Figure 2.7: SDN Architecture

Figure 2.7 [34], represents an SDN Architecture with elements such as an SDN controller, control, and data plane, network applications and devices, and northbound and southbound interfaces. The control plane in SDN is software-based which also offers more flexibility than standard networking. It manages how data packets are forwarded. The data plane physically handles the traffic depending on the control plane's configurations, that is it forwards the packets. Additionally, the centralized software-based controller offers an open interface that allows the administrators to control networks, alter configuration settings, and others from a centralized location. This interface could be the interface to the applications, northbound, or it could be the interface to the devices, southbound.

The northbound interface of the controller gives an abstraction of the network layer and topology, as well as the network protocols themselves. In addition, REST or JSON are commonly used in the northbound API which developers are familiar with. In this northbound API, applications can quickly be connected to the controller and make network

changes. In the case of the southbound API, the controller uses the OpenFlow interface (see Section 2.5.2) to manage the network devices.

These SDN applications can define flows on a device, tell them how to respond to a packet forwarded to the controller, redirect traffic for purposes of inspection, authentication, or related security tasks [4]. So, the need to secure the controller is extremely high since it can help maintain the network secure.

The following sections will mention the most used SDN controllers with a special focus on their security proprieties and support for AAA services.

2.4.1 Open Network Operating System

ONOS has initially been released as an SDN operating system for service provider networks [35] with careful attention to high availability, scalability, and performance.

Recently, ONOS added the support for TLS and added authentication mechanisms to restrict access to ONOS REST API, Graphical User Interface (GUI), and Command-line Interface (CLI). [36].

- ONOS CLI - To access ONOS CLI, there is a need for a public/private key authentication.
- ONOS GUI - To access ONOS GUI, there is the need to fill a form login.
- ONOS REST API - To access ONOSREST API credentials must be given to perform a basic authentication.

ONOS uses the Apache Karaf framework [37], where the Apache Karaf authentication realm is also used by the ONOS CLI, GUI, and REST API. At the moment, ONOS does not support role-based authorization.

2.4.2 OpenDaylight

OpenDaylight is a modular open-source platform hosted by the Linux foundation for scale-out: customizing and automating networks of any size and scale [8]. This modular platform is based on Kafka container, which permits the module to be turned off/on without hindering other modules.

The SDN platform in OpenDaylight is supported by different protocols such, OpenFlow, OVSDB, NetConf, BGP, and many more. OpenDaylight uses Apache Shiro Java security framework [38] to provide Cryptography and Session Management and AAA services, which include Authentication, Authorization, and Accounting.

To support the authentication & authorization schemes, the AAA module in OpenDaylight uses the Shiro Realms [39]. This AAA module has the role of checking the authenticity before allowing communication with the network or the modification of any data. Only authenticated users get access to the resources of the network.

One of AAA module authentication schemes requires a token-claim based authentication [40] . First, the user does a basic authentication in the controller and if the authentication is successful it receives an access token. With this access, it can access the

protected resources on the controller (In the recent releases of OpenDaylight, this authentication scheme was removed). OpenDaylight also has role-based control so access to a restricted resource is only allowed if the user role has enough permissions. With the default configurations of OpenDaylight, there are two roles, *an "admin" role, and a "user" role*. In OpenDaylight, a grant is created when we associate a user with a role. A user with the admin grant can do all available operations. A user with the grant *"user"* has more limitations in the operations it can make, so it can only perform certain operations if it receives an *"admin"* grant. In addition, OpenDaylight by default, in their AAA module, already has a domain called *"sdn"*.

The AAA module records access requests made by a user or an application as part of their accounting method, through the standard slf4j [41] logging mechanisms that are used in OpenDaylight.

In addition, OpenDaylight uses H2, a Java SQL database [42] to save their data. H2 works in an embedded mode in OpenDaylight, which offers a faster and easier connection.

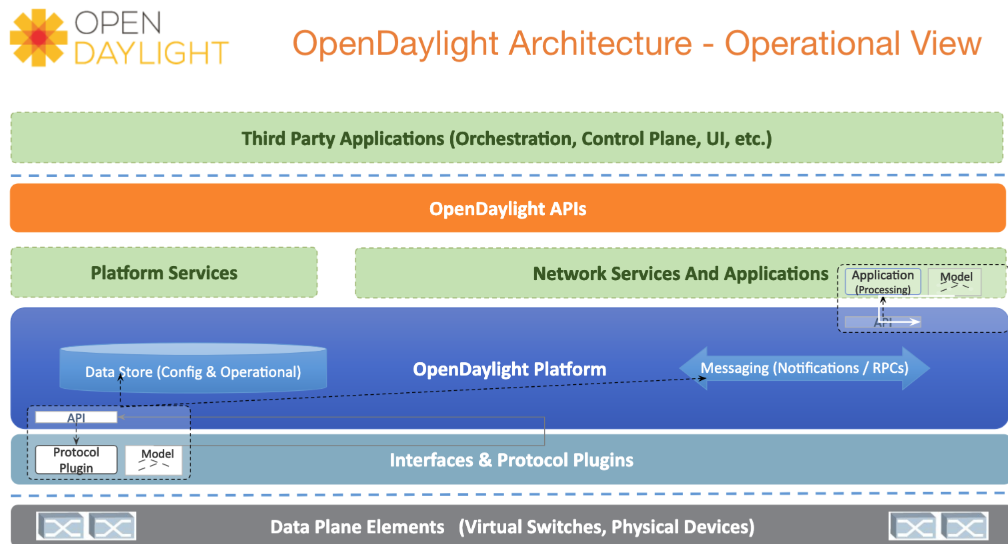


Figure 2.8: OpenDaylight Neon Architecture

Figure 2.8 [43] shows the architecture of the version Neon of OpenDaylight, where the AAA services, which are not visible in this Figure, belong to the platform services.

2.4.3 Lighty.io

Lighty.io is a kit of several OpenDaylight components that supports and eases the development of SDN implementations [44]. Nonetheless, the removal of the Karaf dependence gave more freedom to the framework of choice, since Lighty.io runs in a plain Java SE environment. Lighty.io is also a modular system, which has a core module, as well modules with northbound plugins and southbound plugins. Each application can start and stop these modules independently.

Some components of Lighty.io for the SDN backend are:

- YANG Tools - Yang is a data modeling language used to model configuration and define data. Yang Tools is a group of tools and libraries that give support to Java applications, helping them abstract and process the Yang language.

- MD-SAL - Is a message bus for data and interface models, while providing messaging and data storage capabilities. The MD-SAL uses YANG as the modeling language for these models and offers messaging for services based on YANG modeling.
- Northbound plugins
  - Restconf
  - Netconf
- Southbound plugins
  - Simple Network Management Protocol (SNMP)
  - Netconf

In terms of security, Lighty.io equally adopted the AAA project [45], which brings the same advantages and disadvantages associated with the OpenDaylight controller.

2.4.4 Open Network Automation Platform

Open Network Automation Platform (ONAP) is a platform that automates deployment and management of Virtualized Network Functions [7]. ONAP also offers a project focused on theSDN controller, SDNC, which controller is based on OpenDaylight.

ONAP also incorporates some of AAA services since ONAP's SDNC uses a controller based on OpenDaylight. Even though ONAP uses AAA services, there is no reference to the service of Accounting. However, the version of the OpenDaylight incorporated is the aluminium-SR1 where the AAA project only supports Basic authorization header [46].

One of ONAP's SDN-C subprojects is the SDN-R. The SDN-R release Honolulu [47] offers an improvement in user management since there was a need to upgrade the insecure basic authentication of OpenDaylight for Restconf, to an implementation that includes JWT. To solve those problems, Keycloak [32] was chosen as the new user management system for ONAP. Keycloak provides a GUI and can also act as an IdP or OP, which allows the adoption of OIDC and consequently the use of JWT.

This related work of ONAP with Keycloak is of extremely relevance to our work since the modifications made within ONAP to integrate OIDC and JWT are similar to the modifications needed to integrate OIDC in OpenDaylight.

2.4.5 Ryu

Ryu is an open-source implementation of a SDN controller. It provides software components and APIs that facilitate the control over applications and the creation of new network management. Ryu supports numerous protocols in the southbound interface, including OpenFlow, Netconf, and others. Ryu also has an interesting single process and multi-threaded architecture design [40]. Each service and application will act on different threads of the same core process.

Even though Ryu is well known in the research community, it still lacks essential layers of security. Without an authentication mechanism in the northbound and southbound interface, every application can send OpenFlow messages, and the risk of spoofing is highly increased [40]. In the northbound applications, the threat of identity repudiation is also noticeable with no logging stage.

2.4.6 Comparison

From the controllers presented, the following table summarizes their ability to support the different AAA mechanisms.

Table 2.2: Comparison Of AAA Support Between Controllers

Controllers	Authentication	Authorization	Accouting
ONOS	Yes	No	No
OpenDaylight	Yes	Yes	Yes
Lighty.io	Yes	Yes	Yes
ONAP	Yes	Yes	N/A
Ryu	No	No	No

Only OpenDaylight and Lighty.io fully support the different mechanisms of AAA. ONAP has support for Authentication and Authorization since it integrates some OpenDaylight components. ONOS only supports basic authentication. Ryu since it's a more research-driven SDN controller, at the moment, does not present any support to AAA services.

Following the analysis of each controller and the comparison of different controllers, OpenDaylight seems to be the most promising controller, as well as having the most detailed documentation. Therefore, OpenDaylight is going to be the central piece of this work, with the use of its SDN controller and AAA module.

2.5 Software Defined Network & related protocols

The following SDN protocols are most used to control the packets received by the switches, and devices, as well as ease the communication of the commands between the controller and the different network devices.

2.5.1 REST interfaces

REST is a software architecture style that defines a set of methods for developing a web API. The client and server REST implementations may be done individually without knowing about one other, allowing each component to growing independently, as long as one side knows what type of messages to send to the other. Therefore different clients can hit the same REST endpoints and do the same activities while using a REST interface. This is one of the reasons SDN controllers have REST APIs, where SDN applications can hit the REST endpoints to retrieve a resource in an SDN controller. REST's main advantage is statelessness, which means the server doesn't save information about the client. Instead, each client request contains all of the information needed by the server to process it. REST APIs benefit from these limits in terms of scalability and speedy performance [48].

REST APIs use HTTP requests to perform common database activities including creating, reading, updating, and deleting records (also known as Create, Read, Update and Delete (CRUD)), for example, on a SDN controller. Metadata, authorizations, unified resource identifiers (URIs), caching, cookies, and other information are all included in request headers and parameters in REST API calls [49]. REST APIs employ request and response headers, as well as standard HTTP status codes (e.g., 200 OK). Payloads are used to send and receive data that is too big to fit inside a message. The payload can be in HTML, JSON, or XML format. In an authorized request, for example, a bearer or basic

token must be sent in the Authorization header, and a JSON payload, needed to complete a specific action, can be sent in the request body.

2.5.2 OpenFlow

In classic SDN, the OpenFlow protocol is the standard southbound protocol. It specifies how the SDN controller and the switch/device communicates. The OpenFlow protocol must be supported by any device that wishes to communicate with an SDN controller.

For example, a flow table abstraction is presented by a switch: and each table contains entries that map packets to actions (can be dropped, modify, or others) [4]. If a switch receives a packet and it does not have any matching entry in its flow tables, then it sends that packet to the controller in a message [50]. The controller sends the operations (a set of actions to the flow tables), for the packet received back to the switch through the OpenFlow protocol. The switch adds an entry for these actions and this packet in the table. This allows network managers to split traffic, control flows for maximum performance, and begin testing new configurations and applications.

Finally, if the OpenFlow communications are configured in a TLS connection an extra layer of security is provided, preventing spoofing and DoS attacks on the controller and/or network.

2.5.3 P4

P4 is an open-source programming language designed to tell networking devices, such as switches, routers, NICS, etc, how to handle incoming packets. Its purpose is to serve as an interface between the controller and the network devices. The language itself is declarative and has a syntactic structure akin to C. For efficient execution, the language is designed to be compiled into run-time format [4].

P4 has a few characteristics that show how it can help improve make network decisions on different devices.

- Supports matches and action - Matches represent the matching tables that define what the arriving packets will be compared against. Actions are the action tables that specify what should be done with the packet when it has been matched.
- Reconfigurability - Once switches are deployed, programmers should be able to alter the way they parse and process the packets.
- Protocol independence - Switches should not be bound to any network protocols.
- Target Independence - Any program written in P4 is independent of the underlying hardware

P4 can be used as a programming language for security middleware, while being used to generate actions, tables, and rules, operating as a type of firewall [51].

2.5.4 NetConf

NetConf was developed in the last decade to replace protocols like SNMP, which were mostly used to monitor networks rather than to configure them. Netconf is designed

for network configuration, but it can also be used to manage network faults. Netconf is increasingly being used as the southbound API for SDN applications [4].

The adoption of XML, an XML-encoded Remote Procedure, brings many advantages such as the flexibility of constructing data structures, the availability of free toolkits and APIs, human readability, and the convenience of transport over existing secured channels [52].

Netconf has different attributes that help stand out against SNMP or other protocols [4]:

- Security - Netconf operates over a secure channel.
- Organization - Netconf separates configuration and operational data.
- Announcement of capabilities - The device supporting a Netconf server, in its first interaction with a client, announces all the YANG models it supports.
- Operations - Netconf procedures are Remote Procedure Calls, which can be used by the SDN controller to pass a set of parameters.

2.6 Summary

This chapter introduced security features such as OAuth 2.0, OIDC, SAML and AAA. The different entities of OAuth 2.0 and OIDC were presented, as well as the OAuth 2.0 Authorization code grant and the OIDC Authorization Code Flow respectively. In both methods, an access token is received in the final step of the Grant or flow to be able to access restricted resources. OIDC provides the ability to extend OAuth 2.0 to identify a user and therefore have login and profile information about the logged end-user.

In SAML subsection, all the actors present in the interactions were detailed as well as the explanation of the concept of cross-domain web single sign-on. At last in the AAA subsection, the importance of having AAA services in the network was described, to prevent access to users not allowed in the network. Additionally, AUSF is a portrait of AAA services in the 3GPP 5G Architecture.

Five SDN controllers were also presented in this chapter, as well as their support for AAA services. OpenDaylight, was originally the platform with more support for AAA service and it served as a foundation for other implementations in other controllers as mentioned, such as Lighty.io being a modular system of OpenDaylight components, and ONAP having a controller based on OpenDaylight. With Table 2.2, it was showed that OpenDaylight and Lighty.io have full AAA support, where Ryu does not support AAA services at al.

In the last section of this chapter, it was described three SDN protocols, OpenFlow, P4, and Netconf, and REST interfaces. These SDN protocols help with communication between devices and controllers (OpenFlow), the applicability of the logical decisions in the network (P4), and therefore a better organization of the latter (Netconf). Consequently, with a better grip on what decisions to take depending on the packets received or the state of the network, it will lead to an optimized, organized, and more secure network.

Additionally, in this chapter, related work regarding security features in SDN controllers and web implementations were presented. The OAuth 2.0 related work was more aligned with our work, where OIDC and SAML related work was more focused on web implementations and security vulnerabilities. The research around SAML, also showed that SAML might not be the best protocol in terms of security or privacy, which supports our work regarding the choice of OIDC as the protocol to be used.

From the related work presented, is noticeable the shortage of related work regarding integrations of security features such as OAuth2.0, OIDC and SAML in SDN controllers. SDN controllers are extremely vulnerable in most cases presented however, there is no improvement or increase in research work regarding these issues. This lack of research can serve as a motivation for our work.

Regarding the related work of AAA, is not common to see research work on its use on web integrations when compared to OAuth 2.0, OIDC or SAML since it is a framework for network management and security. The use of AAA can be seen within SDN controllers, such as OpenDayLight, ONAP and Lighty.Io as previously mentioned.

Chapter 3

Research Objectives and Approach

This chapter describes the main objectives of this work as well as the methodology used and the decisions made, before presenting the architecture of our work and the respective workflow. In Section 3.1 we present the research methodology used and in Section 3.2 we start by identifying and describing the research objectives. Following, in Section 3.3, the methodology of this work is presented in detail to explain the procedures needed to accomplish the objectives listed, including the justifications for some decisions made in this section. At last, in Section 3.5, the architecture of our work and the workflow are shown, including the interactions between the different entities.

3.1 Research Methodology

This dissertation started with an exploration of research papers, journals, and more in websites such as IEEE Explore [53], Research gate [54], Elsevier [55], Academia [56], and Google Scholar [57], summarized in Table 3.1. On these websites, there was a search around the keywords of SDN, controllers, authentication, authorization, OAuth 2.0, OIDC and security. Ten articles were found in total, which were reduced to five articles since these were the only ones with relevant information regarding SDN controllers and their security features. The search realized gave insight regarding the SDN controllers most talked about in terms of containing authentication mechanisms as well as other security features.

Table 3.1: Research Inclusion Criteria

Criteria	**Values**
Website	IEEE Explore, Research gate, Elsevier,Academia, Google Scholar
Date	2012-present
Concepts	controller, SDN, security features
Keywords	SDN, controller, authentication, authorization, OAuth 2.0, OIDC and security
Combination	SDN OIDC, SDN OAuth 2.0, SDN controller authentication, SDN controller authorization, SDN controller security

With the SDN controllers found on the articles, which are mentioned in Section 2.4, we start analyzing the official documentation of each, if existing. This analysis consisted in understanding the community support these implementations had, if they were open-source, if their documentation was rich, if they were still producing new releases, as well as understanding the components and security features of the most recent release presented.

For each SDN controller, which presented some security features, such as OIDC, OAuth 2.0, or AAA services, we verified how these features were being used and implemented. In the case of OIDC support, we verified how the integration was made and which OP was used, recurring to their official documentation and if needed, we examined the source code if available. For some controllers, we also tried to run their official code. In the case of using OAuth 2.0, we searched for the use of bearer tokens or basic tokens also in their documentation or source code. At last, if only AAA services were specified, we searched for which authentication and authorization mechanism were being used, consulting their documentation and source code.

3.2 Research Objectives

The main goal of this work is to design and evaluate a framework with authentication and accounting mechanisms in SDN controllers, in order to authenticate end-users accessing SDN resources. To fulfill this objective, this was divided into the following sub-objectives:

1. **Evaluate the impact of the additional parameters with context information in OIDC**.

   First it is necessary to understand how to manipulate the workflow in OIDC to include additional parameters related to a user's device context information in the claim field. The resultant workflow shall be first assessed in a web scenario since most of the OIDC integrations tend to be in a web context. Secondly, it is necessary to evaluate different context values which will result in different levels of trust. From this analysis, we should be able to understand the impact of the addition of context information in OIDC.

2. **Enable the support of OIDC for authentication and authorization in SDN controllers**

   Since we want to design a framework with authentication and accounting mechanisms in SDN controllers, it is essential to first integrate OIDC in an SDN controller, OpenDaylight, to offer such mechanisms. For this integration, there is some analysis needed regarding the workflow, the usage of filters in OpenDaylight, which OP to use, and much more.

3. **Evaluate the impact of OIDC support in SDN controllers**

   We shall assess the impact on both performance and security of OIDC support in SDN controllers.

4. **Enable the support of OIDC with an extension for additional parameters with context information in an OpenDaylight controller**

   The first two sub-objectives need to be accomplished in order to complete this sub-objective. With a framework with authentication and accounting mechanisms in SDN controllers, and a prepared modified workflow in OIDC to include additional parameters, the only thing left is to combine them. To make this combination work, another analysis must be made, regarding the OP to use, to support both context information with a modified workflow in OIDC and a communication with a SDN controller.

5. **Evaluate the impact of OIDC support with an extension for context information in OpenDaylight**

We want to evaluate the impact resulting from a OIDC support in an SDN controller with an extension for context information, in both performance and security compared to a standard integration of OIDC in SDN controllers.

3.3 Approach

This section presents the approach taken to accomplish some of the sub-objectives mentioned. For the sub-objective regarding the addition of context information in OIDC, Section 3.3.1 shows how the context information is added in the claims of OIDC, as well this information is verified in a OP. Section 3.3.2 describes which entities are going to be involved to provide OIDC support in OpenDayLight controller, as well the new flow for authentication in OpenDaylight and the authorization mechanisms used, and the AAA filters that will be created. Finally, Section 3.3.3 explains some of the difficulties that were encountered to integrate OIDC with context information in an OpenDayLight controller and the need for a further investigation of this combination.

3.3.1 Evaluate the impact of the additional parameters with context information in OIDC

As mentioned in the first sub-objective, we will extend OIDC to include context information regarding a user's device in order to provide different levels of trust depending on the level of trust that the user has in the context associated with a device. For example, a personal device used to authenticate an application in the private home network of the user will have more trust and will be considered with a higher level of security, in comparison to the context when the personal device can be used in a public WiFi network.

OIDC besides defining standard claims allows the specification of additional claims, to define these different levels of trust, Table 3.2 presents the fields that characterize the context information of a user's device and will be specified as additional claims in OIDC. Some of these values are represented by string values, others are represented by Universally Unique Identifier (UUID)v5 values. UUID is used to generate unique identifiers, which facilitates the association of user-assigned names with unique identifiers, and the use of v5 is justified by its increase of security since this version uses Secure Hash Algorithm (SHA)-1 algorithm to encode the strings.

Table 3.2: Trust Context Information Fields

Field	**Required**	**Description**	**Values**
deviceID	Yes	Identification (ID) of physical device	UUIDv5
appID	Yes	Identification of application or UA	UUIDv5
appEnvType	No	Type of environment where UA/app runs (e.g., OS, k8s, docker, VM)	String
serviceID	Yes	ID of service with user interaction	UUIDv5
networkID	No	Identification of network	UUIDv5
networkType	Yes	Type of network	{Trusted, unTrusted}

Starting with a deviceID, this field represents the ID of a physical device. That is, if the device used by the end-user is called *"Galaxy S20"*, then, *deviceID* will take the UUID value of the encoded string *"Galaxy S20"*. The field of *appID* represents the ID of an application or a UA. If the value of the UA is *"google chrome"*, the *appID* will have the

UUID of this string. In the case of *appEnvType*, the value of this field corresponds to the type of environment where the UA or the application is running. It could be the case of applications running in Docker or Virtual Machines (VM) environments. The *serviceID* is the UUID of the name of the service being used by the user and *networkID* is the UUID of the name of the network the user is connected to. At last, the *networkType* is the type of network, that is, if we are dealing with a private or trusted network, the network type will have the value of *Trusted*, while in the case of a public or non trusted network, the value would be *unTrusted*.

The table also presents the mandatory fields of context information, which are the deviceID, appID, serviceID, and networkType. These four fields give a general representation of the context of a user's device, therefore being mandatory.

Therefore the inclusion of these claims requires a few changes in a normal workflow of OIDC authorization code flow. The first change is in the initial request to the OP, where the user must also pass along these new fields with the respective values. When the request reaches the OP, besides the standard verification of credentials and specific parameters of OIDC, there must also be a verification of the context information, in order to establish the level of trust for this authentication.

The output of the Algorithm 1, which verifies the context information, enforces the policy to apply according to the level of trust. The values of context information will be compared with the values of context that were registered in the OP upon the creation of an RP. The algorithm starts by verifying if all the mandatory context information claims are present in the request and if those values have an exact match with the values registered in the OP, and the network is trusted, then it will be set a policy *pAll*. This policy will make the inclusion of all 25 available userinfo fields in the ID token [17] (name, email, gender, phone_number, street address, postal code,...). If only the *deviceID* and *appId* have an exact match, but the value of *serviceID* is null, then we will have one of two outputs. If the network type is trusted, the policy set is *pLessTrust*. This policy will make the inclusion of 9 userinfo fields in the ID token (sub, name, email, formatted, street address, street address, postal code, country, locality, region). If the network type is *unTrusted*, then the output is the policy *pUnTrust*. This policy will only make the inclusion of 3 userinfo fields in the ID token (sub, name, email). If either the set of mandatory claims is empty or if the policy is still null at the end of the algorithm, then the authorization request is rejected. If the dictionary with the context information received in the request, is null, that means, the authorization request is a standard OIDC authorization request, and the policy set is *pOIDC*. This policy will make the OP take the standard decisions regarding an authorization request.

Following the successful verification of the different parameters, the OP will present the consent page to the user and normally resume the workflow of OIDC. The ID token will include the fields of user information that were decided with the policy set in the context verification.

So the context information claims will result in the type of user information, Personal Identifiable Information (PII), an ID Token will contain according to both the context information received and the policy set. Since these claims are included in requests of the OIDC workflow, we must encrypt the context information whether with the use of SHA-2 or AES algorithms, so that only the OP can decrypt and understand the values of such claims.

To design and evaluate the new workflow, first in a web context, it is necessary to choose a service to integrate with OIDC. The service will be TeaStore [9] and the implementation

Algorithm 1: Policies Set In The Verification Of Context Information

Input: Dictionary D with context information claims received in the first request
M mandatory fields
Output: Policy P

```
1  P=null
2  if D isNotEmpty then
3  |  if len(M) == len(MandatorySet) then
4  |  |  if D[deviceID] and D[appID] hasExactMatch then
5  |  |  |  if D[serviceID] hasExactMatch then
6  |  |  |  |  if D[networkID] or D[appEnvType] hasExactMatch then
7  |  |  |  |  |  if D[networkType] == Trusted then
8  |  |  |  |  |  |  P=pAll
9  |  |  |  |  |  end
10 |  |  |  |  end
11 |  |  |  end
12 |  |  |  if D[serviceID] == null then
13 |  |  |  |  switch networkType do
14 |  |  |  |  |  case Trusted do P=pLessTrust;
15 |  |  |  |  |  case unTrusted do P=pUnTrust;
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 end
21 else
22 |  P=pOIDC
23 end
```

of a OP and RP will be supported by the AuthLib [31] OpenSource Python library. The service and OpenSource project that was chosen is justified in Section 4.

3.3.2 Enable the support of OIDC for authentication and authorization in SDN controllers

In the following sub-objective, to integrate OIDC in the OpenDaylight controller, we first need to choose the OP. Keycloak is going to serve as the OP in this implementation since it can support OIDC and it has very complete user management. In Keycloak, it is possible to give users roles and set their credentials and their user attributes. It is also possible to create a realm [58], in this case, with the same name as the default domain of OpenDaylight *"sdn"*. A OP supported by AuthLib was also considered, however it does not offer already a complete GUI interface for user management and does not include the support for users roles. It is possible to implement features with OauthLib, however, it would take more time.

So, if there is a OP, it is also necessary a RP in the controller in order to communicate and exchange information with the OP. A class in the OpenDaylight controller must be created to handle these interactions with Keycloak. This OIDC must be registered in Keycloak.

The second modification needed is in the database of OpenDaylight since, at the end of a OIDC flow (recall Authorization code flow in OIDC, Section 2.1.2), the RP will receive the access token, the expiration time and issue time of the respective token, the refresh token, and ID token of the authenticated user. So, the database, which is an H2 database (Java

SQL database), needs to be modified to also support these new fields.

The version of OpenDaylight that will be used, *Silicon* (v0.14.4), supports two authorization mechanisms, Shiro-Based Authorization [38] and MDSAL-Based Dynamic Authorization (recall mention of MDSAL in Section 2.4.3). Shiro uses the *shiro.ini* URLs, where to each endpoint defined in the configurations, a filter can be created and specified to handle the requests of that endpoint. For the same endpoint, the roles of a user allowed to make a request are also indicated. This filter can be coded and therefore we have better control over the way the filter deals with the different requests. For example, Figure 3.1, indicates that all requests made to `/**/operations/cluster-admin**` will be handled by the *authcBasic* filter. Following is also specified that the only role allowed to that endpoint is the role "admin".

```
<urls>
    <pair-key>/**/operations/cluster-admin**</pair-key>
    <pair-value>authcBasic, roles[admin]</pair-value>
</urls>
```

Figure 3.1: Shiro-Based Authorization

MDSAL-Based Dynamic Authorization is similar but it uses the *MDSALDynamicAuthorizationFilter* to restrict access to the resources endpoints. The users with suitable roles can define the list of policies with a simple request. Figure 3.2 [39], presents the request with an updated list of policies to limit access to the different OpenDaylight endpoints. In this example, only users with the role of "admin" can access the resource `/Restconf/modules/*` and perform one of the operations described.

```
HTTP Operation:
put URL: /restconf/config/aaa:http-authorization/policies

headers: Content-Type: application/json Accept: application/json

body:
  { "aaa:policies":
    { "aaa:policies":
      [ { "aaa:resource": "/restconf/modules/**",
        "aaa:permissions": [ { "aaa:role": "admin",
                               "aaa:actions": [ "get",
                                                "post",
                                                "put",
                                                "patch",
                                                "delete"
                                              ]
                             }
                           ]
        }
      ]
    }
  }
```

Figure 3.2: MDSAL-Based Dynamic Authorization

So, our work will use Shiro-Based Authorization for the filter in the controller, and it will use features of Keycloak that are similar to MDSAL-Based Dynamic Authorization regarding the resources, associated with policies. Therefore, a new Shiro-Based Authorization filter must be created on the controller to deal with the requests of this new integration. This filter will sort and process the requests, according to the parameters sent in the request.

- If a request comes without any credentials or tokens, the response will ask the user

to provide at least one of the two.

- If a request has some credentials, such as username/password, the filter will understand that the user is not authenticated since the user did not provide a bearer token. It will, therefore, redirect the request to the RP class, to start the authentication process.
- If the user provides a bearer token, the token in the header of this request is extracted and validated. If valid, then the filter will allow the user to access the endpoint specified in the request.

The architecture and the workflow of this integration are explained in detail in the following Section, 3.5.

3.3.3 Enable the support of OIDC with an extension for additional parameters with context information in an OpenDaylight controller

For this sub-objective, to replace the standard workflow of OIDC in the second sub-objective with the modified OIDC workflow which includes context information, it is necessary some modifications in the OP previously chosen, Keycloak. One of these modifications includes the extension of the database of Keycloak to register fields related to context information.

To integrate the modified OIDC workflow, the user in the authentication request, besides its credentials, must also provide the context information in the request body.

3.4 Use case

A use case for the work proposed is the administrative activities in the network that need interaction with the OpenDaylight controller. As different operations may require different levels of authority, we need to differentiate those levels and therefore have fitted roles. As seen from the state of the art, nowadays, authentication based on passwords is not sufficient to provide enough security for a central component of a network. So a solution for that problem is the use of federated mechanisms, such as OAuth 2.0 and OIDC.

Trust Context Information (TCI) is relevant for the use case when a user needs to perform a sensitive administrative operation. However, the context of the device used for that operation may not allow the operation to occur, if the context is not considered trustable enough.

The automatic detection or verification of the authenticity of the context information, even though it is important, is out of the scope of this work. We will assume that the context information sent by the users during this work is authentic.

3.5 Architecture

In this section, an SDN architecture with an OIDC integration is presented, as well as the workflow between the different entities shown in Figure 3.3. Not all entities in this figure are going to be presented in the workflow of this work, including the network devices

since we will not be working with the Southbound Interface, and the SDN applications, since every request is going to be made directly from the user interaction with the CLI of OpenDaylight.

The first relevant entity is the User, which is the end-user who wants to access the restricted resources of a SDN controller. Following, we have CLI which is the interface used by the user to interact with the SDN controller. Inside the SDN controller we have the AAA filter, which will regulate the authentication and authorization mechanisms. That is, according to the user's roles and credentials, it will decide if the user request is denied or allowed and passed to the RP, or if the user is already authenticated, it will allow direct communication to retrieve or manipulate the restricted resources of the SDN controller. The other component inside an SDN controller is the RP, which is necessary to authenticate the users that do not have valid tokens. On the side of the SDN controller, there is the H2 database, as previously mentioned. At last, Keycloak is the OP of this work, which will communicate with the RP inside the SDN controller in order to authenticate the users and provide the necessary tokens for a user to access the SDN controller resources. Keycloak also has a database.

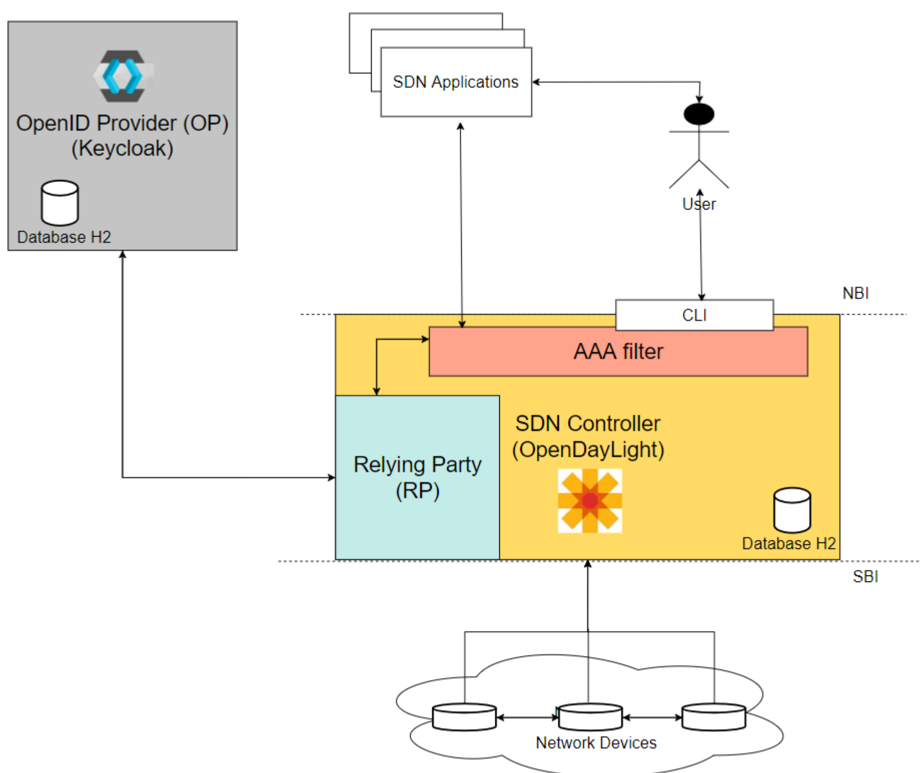


Figure 3.3: SDN Architecture With OIDC Integration

Regarding the workflow, a user starts by making a request directly in the CLI of the SDN controller. This request is directed to the AAA filter, where, if the user provided a bearer token, the token is going to be verified in order to authenticate the user. If the token is valid, the AAA filter just needs to check the user's roles to understand if the user is authorized to perform the intended operation. If the user has the necessary permissions, the operation is performed, and the user obtains a response in the CLI. This operation is also logged since the SDN controller also has accounting mechanisms. If the user does not provide a bearer token but instead provides credentials for authentication, then the request received by the AAA filter will be directed to the RP, which will communicate with Keycloak to authenticate and receive the necessary user's tokens. These tokens are

stored in the SDN controller database for future use. Upon receiving the valid tokens, a response to the client is presented in the CLI with the respective access token, which now allows the user to make requests to access the resources of the SDN controller. In the case where a user does not provide a bearer token in the request or does not give the necessary information for authentication, the AAA filter will decline the request and give a response to the user, indicating that the user needs to give a valid bearer token or credentials in the request (workflow in more detail in Section 5.3).

Chapter 4

Preliminary Results

This chapter presents the activities and the outputs related to the first sub-objective: Evaluate the impact of the additional parameters with context information in OIDC. First, the integration done in both TeaStore, OP and RP will be presented, followed by the evaluation scenario, metrics, configurations, and finally the presentation of the results. The results presented are based on the work achieved in Appendix A.

4.1 Integration

As mentioned in Section 3.3.1 the service chosen to integrate OIDC with is TeaStore. "The TeaStore emulates a basic web store for automatically generated, tea and tea supplies. (...) The TeaStore is a distributed micro-service application featuring five distinct services plus a registry" [10]. However, the authentication in TeaStore is only supported in a form-based fashion, thus not supporting OAuth or OIDC authentication flows. Consequently, many modifications were necessary:

- **Database and classes modified**

  The database in Teastore suffered some alterations since it needed to save the tokens and all the parameters involved in OIDC regarding a user. The user's classes were also changed to support the retrieval and modification of those parameters.

- **Create endpoint for context information**

  In Teastore, there was a creation of an endpoint called "context" to receive the context information regarding a user's device.

- **Create endpoint for token**

  In Teastore, there was an endpoint creation called "token" to receive the OIDC tokens generated by the OP.

- **Password Changed**

  To differentiate the users that were authenticated through OIDC or through a basic form, for the users that chose OIDC authentication, the field of password was changed to "openid".

- **Login**

After TeasTore receives the user's tokens generated by the OP, it stores them in the database and also validates the user's token. If the user has the password as "openid" and if the access token is valid then the user is authenticated and can access TeaStore.

- **Request verification**

  Every time a user loads a webpage, the function `isLoggedIn` is called, where, similar to the previous modification, if the user has the password "openid" and if the access token is valid then the request is valid.

- **Refresh Token**

  When a request is verified and 50% of an access token lifetime has passed, TeaStore will use the user's refresh token to access a RP and request new tokens. The new tokens are also stored once received. It was decided a new token request after a lifetime of 50% since if the token has a short life, and if we do not know when the next request and next verification will come, the access token will rapidly become invalid and consequently, the user will need to go through the process of authentication once more.

All entities that will be mentioned in this section use Hypertext Transfer Protocol Secure (HTTPS) to communicate with each other. Even though TeaStore was not able to support HTTPS, as previously mentioned, one of the contributions of this work was the improvement of TeaStore to support HTTPS communications.

The implementation of the OP and RP was supported by the AuthLib Python library, as previously mentioned. The use of AuthLib was mainly justified since this open-source Python library offers more flexibility to modify if necessary the OIDC flow or other libraries it contains. So, some libraries offered by the AuthLib were modified to support the addition of the context information in OIDC. The RP was modified to include the context information regarding a user's device in the initial request. The context information was also encrypted with a Fernet key [59] to provide data confidentiality. Besides the standard functions supporting an OIDC authentication code flow, implemented in the RP, was also added a function to handle the refresh token, where the client communicates with the OP and exchanges a refresh token for new access and refresh token.

In the OP, the modifications included the addition of the algorithm described in Section 3.3.1 upon the reception of the first authentication request. In this function, the values of context received are decrypted with the use of Fernet library, before processing them in the verification algorithm. To make the comparisons with the registered values of context in the algorithm, the client class in the OP was also changed in order to save the context information registered upon the creation of a RP. Two libraries of OP were modified, where the first one was changed to verify the policy set at the end of the algorithm of context verification, and therefore decide which user's attributes the ID Token will have. To support the 25 user's attributes specified in OIDC (recall the mention of claims about the end-user in Section 2.1.2), the user class was also changed to support and save data of those fields. The second library was changed to include a refresh token alongside the tokens received by the client.

4.2 Evaluation

4.2.1 Scenario

Figure 4.1 represents the scenario of our evaluation for the first sub-objective. The entities of this scenario are a OP, a RP, a TeaStore service with support for OIDC authentication, locust to and ElasticSearch. Every entity is a Docker [60] container, including the individual services of TeaStore. TeaStore, OP and RP were all in the same physical machine, while locust and elastic search were in a distinct physical machine each.

To evaluate our scenario, it was used two tools present in the Figure 4.1, Locust, and ElasticSearch. Locust [61] is a load testing tool, that can swarm web services like TeaStore with many users and requests. It was possible to define a user behavior, where it interacted with TeaStore and OP and RP, in the different steps of the OIDC flow. In locust, we created datasets with users and context information, which correspond to different environments, where the user can interact with TeaStore. Consequently, the context information datasets included the three available policies. In the end, locust presents different metrics related to the requests performed.

ElasticSearch [62], a free and open analytic engine, will be used to monitor the different dockers' containers and retrieve different metrics. In particular, those assessing resource consumption (e.g., CPU usage).

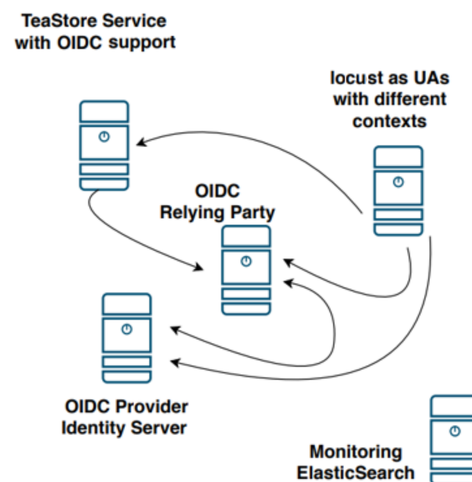


Figure 4.1: Evaluation Scenario

Figure 4.1 also demonstrates the communications between each entity. TeaStore always uses the OIDC client to communicate with the OP. Locust besides making requests to the TeaStore, if needed can make requests directly to the RP and OP, considering the modeling of user behavior in the authentication flows.

Figure 4.2 shows the overall evaluation scenario flow between the different entities, excluding ElasticSearch. Locust can define a user behavior interacting with TeaStore through a User Agent, therefore in this image, only the name of User Agent appears. To simplify the mention of the trust-related with the context information, the acronym TCI will be used throughout the rest of this dissertation.

The flow of our scenario starts by making a POST request from locust to TeaStore, containing the TCI of a user's device. TeaStore upon receiving this information also requests

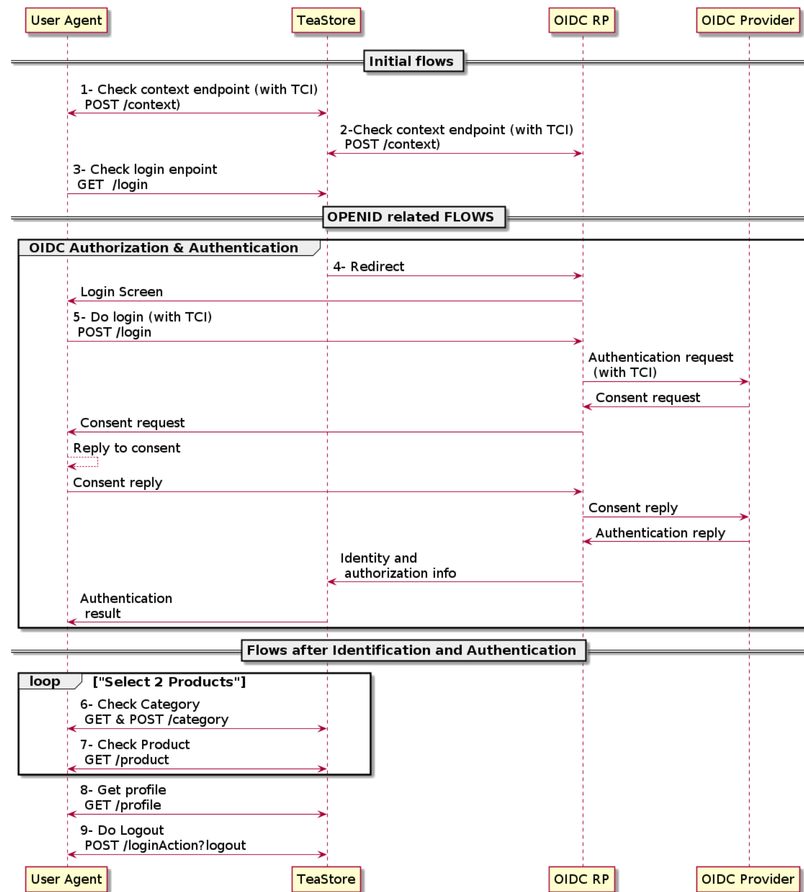


Figure 4.2: Evaluation Scenario Flow

with the same TCI, to the RP. Following this interaction, locust makes a get request to the login endpoint of TeaStore, where the request is redirected to the RP. The RP proceeds to make an authentication request with the TCI previously received, and the OP verifies all the parameters and TCI, responding with a consent request if the verification is successful. The consent request reaches locust, where locust gives a consent reply to the RP. The RP delivers the reply to the OP, where it verifies the message received. If successful, it generates the tokens, access token, refresh token, id token, and other parameters, delivering that information to TeaStore. TeaStore verifies the token received and if valid, the user is authenticated. After being authenticated, locust can make different *Get* and *Post* requests in the TeaStore to complete different operations, ending the flow with a *Post* request to log out.

4.2.2 Configuration parameters

Regarding the configuration parameters for our evaluation, Table 4.1 indicates the parameters and the respective values. For the access token lifetime, three values will be used, 30, 60, and 600 seconds, in order to have scenarios with more refresh token requests and others with fewer requests. The number of users also varies between 10, 50, and 100, to have environments with fewer simultaneous requests than others. Another parameter includes the policy related to the TCI of each user in the datasets that were created. In the end, two scenarios were tested, in order to compare the results. The scenario with OIDC acts as a baseline for comparison, with the TCI scenario that includes a modified flow with TCI on top of a OIDC standard implementation.

Table 4.1: Configuration Parameters

Parameter	Description	Values
T_{life}	Token lifetime	30s, 60s, 600s
n_{Users}	Number of users	10, 50, 100
pol_{ABAC}	Policies to be applied by OP	pALL, pLeTrust, pUnTrust
Scen	Scenario w/ and without context info	OIDC, TCI

4.2.3 Metrics

Table 4.2 presents the metrics collected for the evaluation of the current work. In the OP, two different metrics were collected. T_{Tissue} represents the time of issuing a token in milliseconds after the OP receives a consent reply. $T_{TCIvalidation}$ is the time it takes to validate the TCI in the implemented Algorithm 1. In TeaStore, $T_{Tvalidation}$ is measured in milliseconds as well and measures the time it takes to validate if an access token is still valid. Locust supports multiple metrics, but the ones collected were *SuccessRate*, *ResponseTime* and *ContentSize*. At last, in ElasticSearch was collected the percentage of CPU used and memory usage of the dockers, as well as the number of packets TX-transmitted and RX-received. These monitoring metrics are not going to be presented in the next section - results, since they did not bring any more relevant information.

Table 4.2: Evaluation Metrics

Metric	Description	Measured
T_{Tissue} $T_{TCIvalidation}$	Time to issue a token (ms) Time to validate TCI (ms)	OIDC OP
$T_{Tvalidation}$	Time to validate a token (ms)	TeaStore
SuccessRate *ResponseTime* *ContentSize*	Success Rate of requests (%) Response time (ms) Request Content Size (bytes)	Locust
cpu *mem* *net*	Percentage of CPU Percentage of memory Packets TX and RX	Monitoring ElasticSearch

4.3 Results

Figure 4.3 shows the time in ms for token verification in TeaStore, for different types of token lifetime and a different number of users both in a scenario without context parameters and in a scenario with context parameters. Tokens with a shorter lifetime, 30 seconds, tend to have more refresh operations, so since this refresh process occurs when the token is verified in TeaStore, the time for this operation will be longer, than tokens with a longer lifetime, 600 seconds, since it is not necessary to perform refresh operations so often. In terms of the relationship between OIDC and TCI scenarios, the values obtained are generally similar, since, in this operation of getting new tokens, there is no direct involvement with context information. In terms of user variability, we verify that the greater the number of users performing requests, the higher the verification time obtained is. These values can be justified since, in this operation, different services in TeaStore are involved, Auth service and Web service. During this process, there is also a communication with the database of TeaStore to retrieve the user's tokens.

Figure 4.4 evaluates the issue time of a token in OP and shows whether it is different

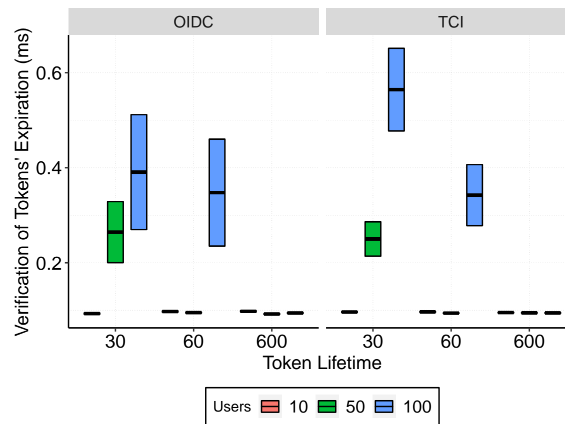


Figure 4.3: Token Verification Time On TeaStore

between OIDC or TCI scenarios. It also shows the time considering the number of users or even different policies, the token issue time relies on an average of close to 40ms. This is expected since the issue part of the token is not related to context information. The issue of a token does not depend on the policy set, as well as on the number of users.

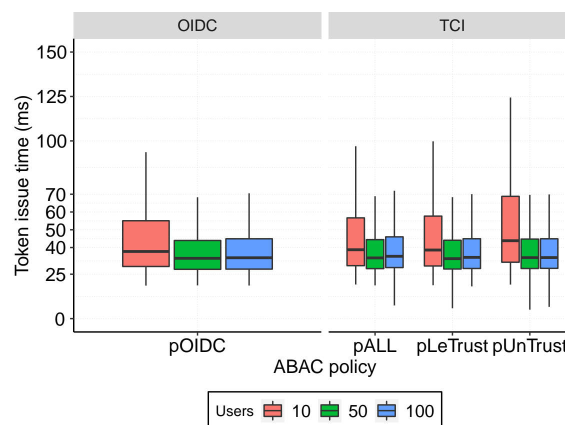


Figure 4.4: Token Issue Time On OP Per Policy

Figure 4.5 shows the context parameters having a significant impact on the verification time of the context information. In the case of the policy for a regular OIDC scenario, this does not include any of the context fields so the time obtained for this operation is close to 0ms. These results obtained for TCI scenarios vary between 4 and 8 ms. Even though the values are high compared to the scenario of OIDC, they are justified by the fact that these context values need to be unencrypted (since when they are sent encrypted from the RP to the OP), and then all the fields have to be checked, whether they are mandatory fields or not. However, between a different number of users, these values do not vary much, since even for a small or higher number of users, this verification for TCI scenarios, will be equal. Between different policies, the values are also similar because the verification of the fields is independent of the values in those fields and consequently the policy set.

The last Figure 4.6 presents the response time per user. This measure corresponds to the time between the request which sends context information from the TeaStore to the RP until TeaStore receives the different tokens at the end of an authentication code flow. Or, in the case of a standard OIDC authorization code flow, from the first request from TeaStore to the RP until TeaStore receives the generated tokens. The most notorious

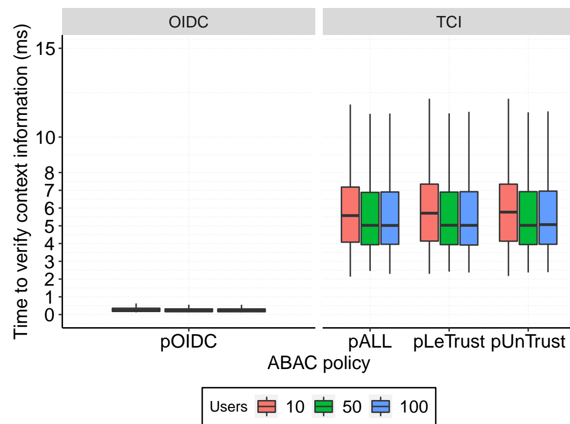


Figure 4.5: Context Verification Time On OP Per Policy

difference is with the rise in the number of users which has a bigger impact than the type of scenario. The number of users impact can also be justified since, during an authentication flow, different services in TeaStore are involved, as well communications with the TeaStore database are established. The small impact introduced by TCI is due to the size of the exchanged messages since they include more information than messages without context variables and the process of verification context in the OP.

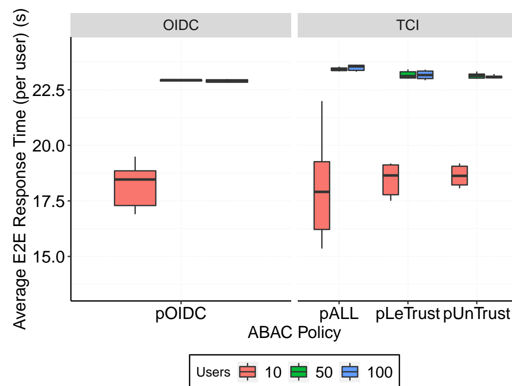


Figure 4.6: Overall E2E Time Per User

All the results presented show that the integration of context information in OIDC does not add a high cost to operations therefore this implementation can be seen as a feasible solution to increase the security in OIDC as well as give the users different levels of trust regarding the context of the device used to authenticate.

4.4 Summary

Despite the promising results, the real benefit of using context information needs further evaluation and more objective metrics. The preliminary evaluation only considers the comparison with the standard OIDC implementation and does not assess the benefits of using context information to protect the assets of users or even assets related to network management.

Chapter 5

Implementation

This chapter presents the work to implement and enable the support for OIDC authentications in the OpenDaylight controller, with and without the extension for TCI. The success of this implementation depends on the help of Keycloak as the OP as previously mentioned, the AAA filter in the controller, and the creation of an RP in the controller, as the communication bridge between the OP and Keycloak.

The modifications and configurations done on the different components of the architecture presented in Section 3.5 and illustrated in Figure 3.3 are described, as well as the flow of the authentication process, verification, and renewal of an RPT token.

5.1 Concepts

The concepts of roles and grants were briefly introduced in Section 2.4.2 but a more detailed explanation is needed to understand how the implementation of this framework works.

5.1.1 Roles and Grants

By default, in OpenDaylight there are only two types of roles: *"admin" or "user"*. Additionally, in OpenDaylight, the definition of a grant stands by the association of a role to a user. When a user is created with the role *"user"*, in OpenDaylight, there is the creation of a user and the creation of a grant, to associate the role *"user"* with the user just created. However, as explained, a user with this grant *"user"* does not contain enough permissions to make requests to restricted resources in the controller. Therefore, it must be given a grant with the role of *"admin"*, to gain enough permissions to access those resources (i.e. restricted resources in OpenDaylight are the resources behind the REST endpoints, so for the AAA module, *"http://ip:port/auth/v1/users"* is the user store in OpenDaylight which is a restricted resource as well as *"http://ip:port/auth/v1/roles"*, the role store).

Nonetheless, **having only two user roles, where a user with the grant "admin" has all the privileges does not meet the security requirements in SDN management use cases.** More detailed and specific roles must be created in the controller. For example, two users, A and B, both have a grant *"user"*. Using the restricted resources presented in the previous paragraph as an example, user A can have an additional grant that only allows it to access the user store, while user B can have a grant that only allows it to access the role store.

Therefore, the idea of having a role for each REST endpoint the controller offers a more refined and secure solution. In the AAA module in OpenDaylight, which is the module we are working, there are **4 REST endpoints**: users, roles, grants, and domains REST endpoint. For this implementation, it was decided that for each endpoint there could be a role called *"grantedUser", "grantedRole", "grantedGrants", "grantedDomains"*, besides the standard role of *user* and *admin*.

This same idea can be applied to other modules in the OpenDaylight. In the Netconf and Restconf module in the controller, there are REST endpoints for topology management, so the existence of a role *"grantedTopology"* would also ensure the security that only the users with this grant or users with the grant of *"admin"* could make requests to the topology REST endpoint.

For example, a new user A was created and associated with the role *"user"*. However, this user wants to access the role REST endpoint (e.g *"http://ip:port/auth/v1/roles"*) even though it does not have enough permissions. In order to gain the permissions required, another user, B, registered in the controller who has a grant *"grantedGrants"*, therefore can access the grants REST endpoint (e.g *"http://ip:port/auth/v1/grants"*), would need to make a request to that grants endpoint, to associate the role *"grantedRole"* with user A. Only then, user A would have enough permissions to access the role REST endpoint in the AAA module.

Eventually, a user with the grant of *"admin"* can access any REST endpoint in the controller, and if there were administrative REST endpoints, it could access them too. However, if we wanted a user to be able to reach any REST endpoint in the controller excluding the administrative REST endpoints, we would need to associate that user with all the existent roles that have permissions to access the different endpoints in the controller. Instead, if we create a role called *"grantedAll"*, a user who receives this grant, would be able to access all REST endpoints with a single role, without having the same permissions as *"admin"*, and therefore without having access to administrative REST endpoints.

Table 5.1 summarizes the role-endpoint mapping suggested.

Table 5.1: Role-Endpoint Authorization Mapping

Module	**Role**	**Endpoint**
AAA	grantedUser	http://ip:port/auth/v1/users
AAA	grantedRole	http://ip:port/auth/v1/roles
AAA	grantedGrants	http://ip:port/auth/v1/grants
AAA	grantedDomains	http://ip:port/auth/v1/domains
Netconf/Restconf	grantedTopology	http://ip:port/restconf/config/network-topology:network-topology/topology/topology-netconf/node/new-netconf-device
All	grantedAll	any
All	admin	any

In the framework presented, there must be a registration of all the roles mentioned in the database of the controller at the initialization of the controller, excluding the role of *"admin"* and *"user"* that is already registered by default. When working with other modules in OpenDaylight, we must also register the roles that restrict the respective REST endpoints at the initialization of the controller, Figure 5.1.

```
h2Store.createRole("grantedALL","sdn","role to grant access to all endpoints");
h2Store.createRole("grantedUsers","sdn","role to grant access to users endpoint");
h2Store.createRole("grantedRoles","sdn","role to grant access to roles endpoint");
h2Store.createRole("grantedGrants","sdn","role to grant access to grants endpoint");
h2Store.createRole("grantedDomains","sdn","role to grant access to domains endpoint");
h2Store.createRole("grantedTopology","sdn","role to grant access to topology endpoint");
```

Figure 5.1: Roles Registration In OpenDaylight

5.2 Components

5.2.1 Keycloak

Keycloak is being used as an OP and a user management interface in this implementation. Therefore, its database regarding users and roles must be synchronized with the controller's database. To maintain this synchronization, a third element, RP, is used to establish a communication between Keycloak and the controller, which is explained in the next section.

Even though in Keycloak, the term grants does not exist like in OpenDaylight, the concept of a grant can be portrayed in Keycloak as the assignment of a role to a user, which portrays the same functionality. So for every role and user that exist in the controller, it must also exist in Keycloak, therefore the necessity for databases synchronization. In future work, the goal is to keep all the user's information only in the Keycloak database, since the user management is performed primarily in Keycloak. The controller would only need to have information regarding a userid and its refresh token, for the token renewal procedure. With the lack of available documentation online for OpenDaylight, it was not possible to make yet this transition, therefore in this implementation, we are working with some duplicates on the Keycloak database and the OpenDaylight controller. The duplicated data consists of users ids and user's tokens (RPT token and refresh token) and roles.

Since there are many configurations in Keycloak, here is the list of steps for a better understanding (recall Section 2.3.2 to comprehend the configurations terms in Keycloak that will be presented in each step):

1. Creation of a realm *"sdn"*.
2. Creation of roles.
3. Creation of the default user with the *"admin"* role.
4. Configuration of an administrative OIDC Client *"admin-cli"*.
5. Creation of a OIDC client to authenticate the controller's users *"controller"*.
6. Configuration of the authorization data in the *"controller"* client.

In this implementation, the **first step** is to create a realm in Keycloak. It was created a realm called *"sdn"*.

The second step is the creation of the same roles existent in the controller, as Figure 5.2 shows. Keycloak by default, upon creation of a realm already creates some roles, therefore the following Figure shows more roles than the ones on the controller. The role *"admin-role"* is a role that was created by us but will be explained in the next step. The role *"authorization"* is also a role created for this implementation and it will be explained in the fifth step.

Figure 5.2: Roles In Keycloak

The third step is the creation of a user called *"admin@sdn"* through Keycloak register form, where we also put some information regarding username, first name, last name, and password. This user is then given the role *"admin"* to match the default admin user that exists in the OpenDaylight controller.

Before advancing to the next step, in this implementation, Keycloak must have at least these two functional OIDC clients, as listed in Table 5.2:

Table 5.2: Keycloak Functional OIDC Clients

OIDC Client	Function
"admin-cli"	User management Operations over Keycloak Admin REST API
"controller"	OpenID authentication

The fourth step shows the configuration of the first OIDC client which is a default Keycloak *"admin-cli"* that is used to retrieve access tokens that give permissions to operate standard user management operations via HTTP request, using the Keycloak Admin REST API [63], instead of using the Keycloak user management interface. However, if this client is used with the default configurations, the access tokens generated do not have enough permissions to create users, delete users, create roles, put roles, and more. These limitations were found over trial and error, and to overcome these problems other sources other than the official documentation were used. As discovered, additional configurations are necessary for this client starting with the creation of a Composite role using the subset of "Client Roles" in the option of composite roles in Keycloak. The created role uses the associated roles from the available roles in the subset of "Client Roles". For example, the creation of a role called *"admin-role"* encapsulates the roles *"view users, query-groups, query-realms, query-users, ..."* of the client *"realm-management"* as Figure 5.3 suggests.

Then, in the configurations of the *"admin-cli"* OIDC client, it is necessary to specify that every user that gets an access token from this client, will have this role of *"admin_role"* included in the token. This attribution is done by assigning this role in the "Scope" and "Service Account Roles" tab of the client configuration, see Figure 5.4 for more details.

The fifth step is the creation of the second client, a standard OIDC client named *"controller"*, with the option for authorization enabled in its configurations. This option will give a similar sense of security as the MDSAL-Based Dynamic Authorization in the controller, in which are resources specified with the type of HTTP operations that can be performed (e.g., GET, POST, PUT, DELETE).

In the last step, we specify the resources for this client *"controller"* which are all the

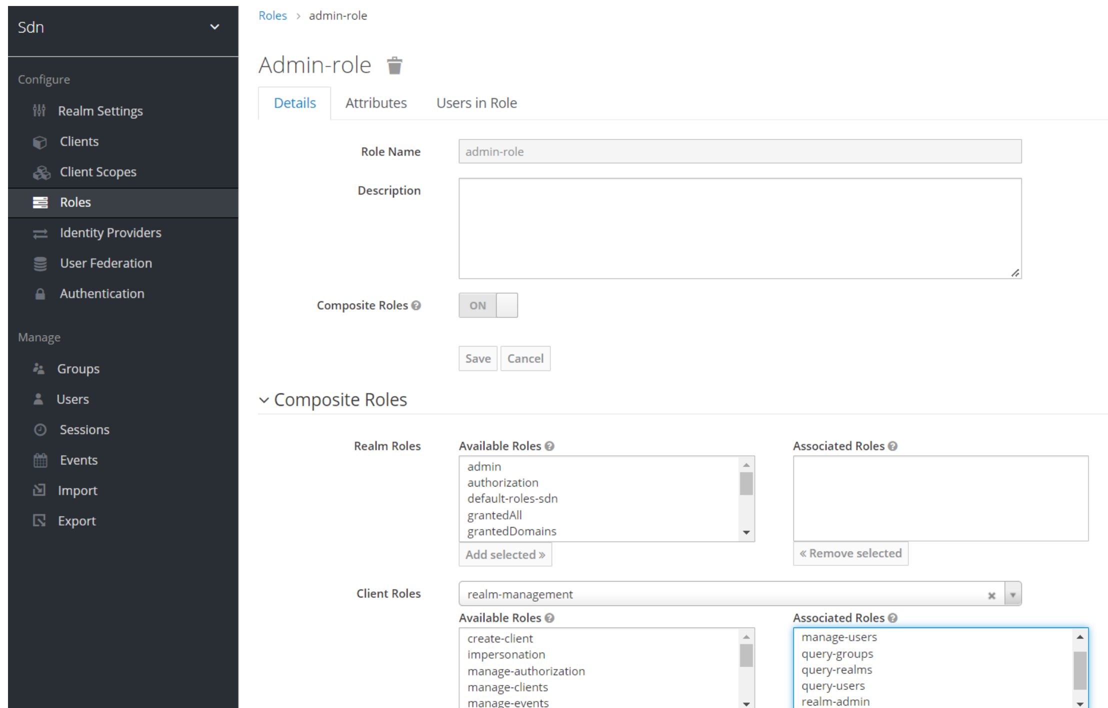


Figure 5.3: *"admin-role"* In Keycloak

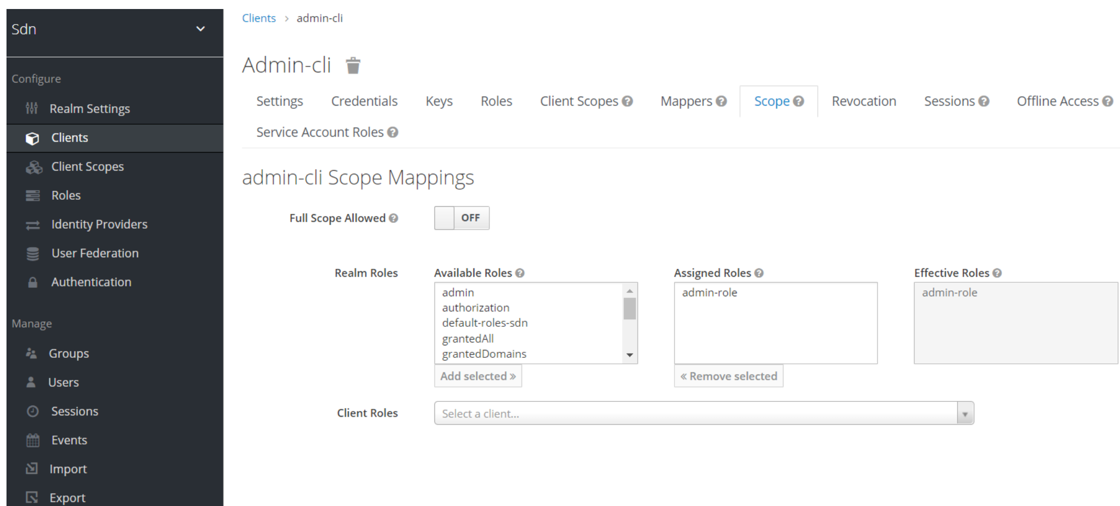


Figure 5.4: Scopes Mappings Of *"admin_cli"* Client In Keycloak

REST endpoints of the controller that we want to restrict, Figure 5.5.

Additionally, it is also defined the resource *"Controlador"*, which indicates the permission associated with the basic access to the controller. A resource of *"Context"* is also defined for the permissions regarding the TCI in the authentication, which will be explained in the following paragraphs.

There are also authorization scopes created that are going to be associated with the resources created, Figure 5.6. So we create the basic four HTTP operations as authorization scopes *get, put, post, delete*. We also create an authorization scope of *trustAverage*, *trustHigh*, and *trustLow*, which correspond to the policy assigned after evaluating the TCI for an authentication and authorization process.

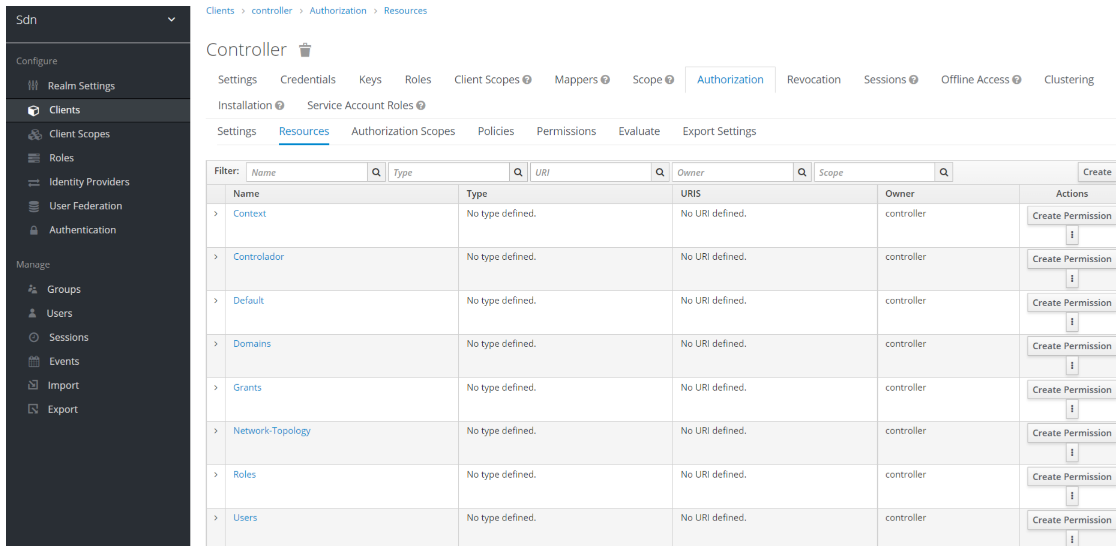


Figure 5.5: Resources Of The *"controller"* Client In Keycloak

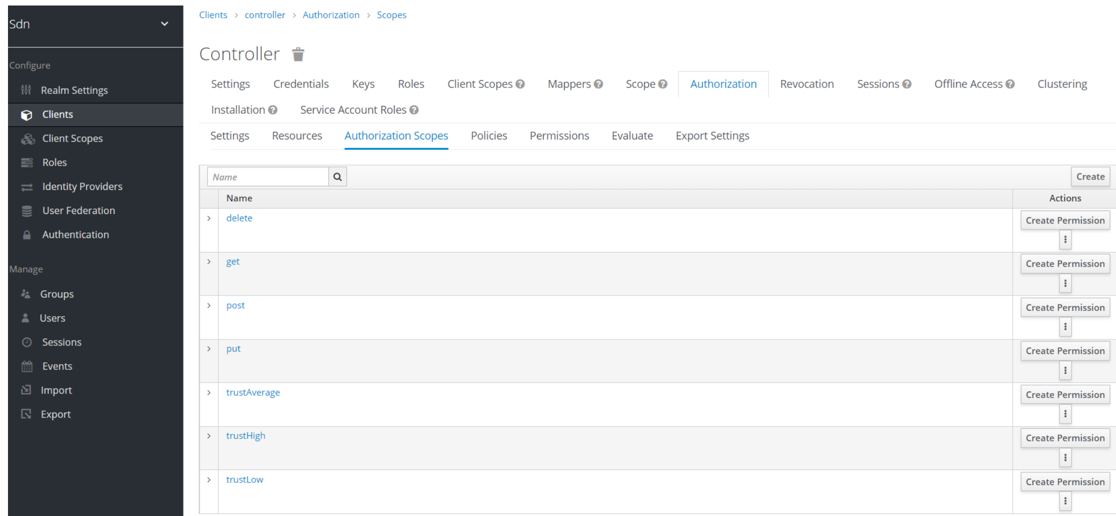


Figure 5.6: Authorization Scopes Of The *"controller"* Client In Keycloak

Following we create policies, and for this implementation, role policies and *javascript* policies. So, for each type of user role it is created a role policy, as depicted in Figure 5.7 (The policy "authorization" is related to the role *"authorization"* given to new users, as the basic role to access the resource *"Controlador"* which is the controller). For the evaluation of the TCI it is also created *javascript* policies, one for the High trust policy, Average trust policy, and Low trust policy. Each javascript policy contains part of Algorithm 1 described in Chapter 4, corresponding to the evaluation of the policy in question. *"ContextHigh"* policy contains line from 1-8 of the Algorithm 1, *"ContextAverage"* policy contains lines from 1-14 1, and *"ContextLow"* policy contains lines from 1-15 of the Algorithm 1.

Finally, we can create permissions, both resource and scopes permissions, as depicted in Figure 5.8. Scope permissions associate the policies created with the authorization scopes and resources.

For example, as in the Figure 5.9, the creation of a scope permission *"scopeDomains"* that associates the resource *"Domains"*, with the different authorization scopes allowed "GET", "PUT", "POST", "DELETE", and the role policies that can access this resource, in this

Figure 5.7: Policies Of The *"controller"* Client In Keycloak

Figure 5.8: Permissions Of The *"controller"* Client In Keycloak

case, *"admin"*, *"grantedall"*, *"grantedDomains"*. For the TCI, it is also created 3 different scope permissions, where for example, the resource *"Context"* is associated with the scope *"trustHigh"* and the *javascript* policy created for the High Trust Evaluation.

Resource permissions associate the resource with the policies created. For example, it created resource permission for the resource of *"Controlador"*, which applies all the role policies of the different types of users existent, Figure 5.10.

Table 5.3 comes in need to summarize all the policies, permissions, scopes, and resources that were configured, since the number of configurations can be overwhelming.

All this work for the authorization is only acknowledged if there is a request of an RPT token with the access token received in the authentication since an RPT is a token equal to the access token, however, includes all the permissions (authorization data) that a user has regarding the resources it can access, as depicted in Figure 5.11.

Finally, the users of this framework will use an RPT token in the authorization header of the requests to the controller since when the decryption of this token occurs, it becomes easier to understand what resources the user has permissions to access and what operations can it do (decryption process explained in more detail in Section 5.3.2).

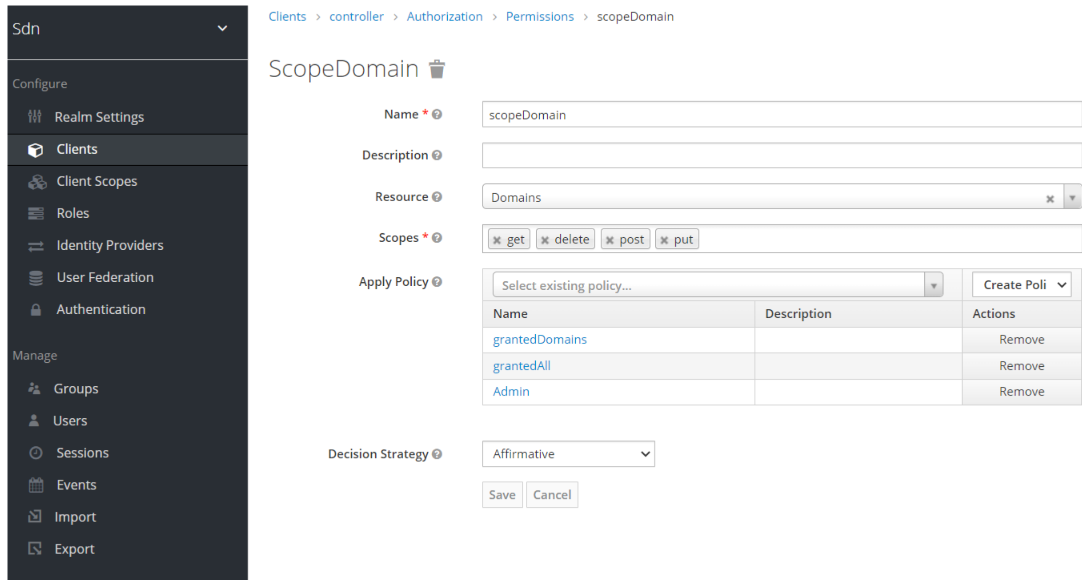


Figure 5.9: Scope Permission *"ScopeDomains"* Of The *"controller"* Client In Keycloak

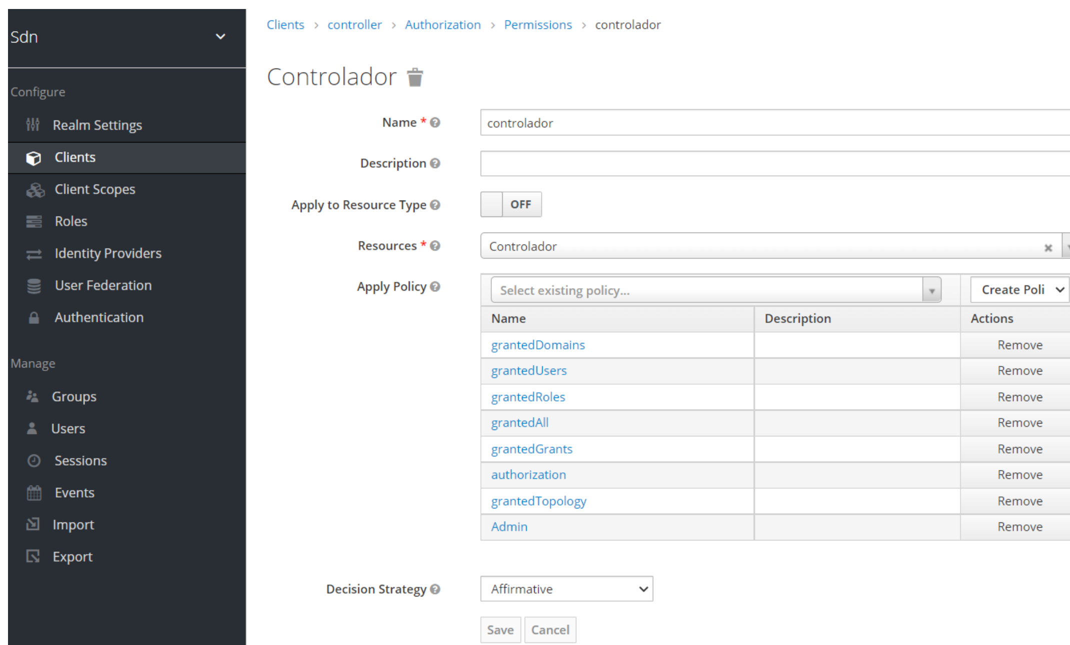


Figure 5.10: Resource Permission *"Controlador"* Of The *"controller"* Client In Keycloak

Table 5.3: Authorization Configurations Summary

Permissions	**Resource**	**Authorization Scopes**	**Policies**
controlador	Controlador	-	grantedDomains grantedUsers grantedRoles grantedAll grantedGrants authorization grantedTopolofy Admin
scopeDomains	Domains	get post put delete	grantedDomains grantedAll Admin
scopeGrants	Grants	get post put delete	grantedGrants grantedAll Admin
scopeRoles	Roles	get post put delete	grantedRoles grantedAll Admin
scopeTopology	Network-Topology	get post put delete	grantedTopolofy grantedAll Admin
scopeUsers	Users	get post put delete	grantedUsers grantedAll Admin
trustAverage	Context	trustAverage	contextAverage
trustHigh	Context	trustHigh	contextHigh
trustLow	Context	trustLow	contextLow

PAYLOAD: DATA

```
{
  "exp": 1655689798,
  "iat": 1655689738,
  "auth_time": 1655689738,
  "jti": "bf179b0e-f920-4cca-a43e-92d8a379d6fa",
  "iss": "http://127.0.0.1:8080/auth/realms/sdn",
  "aud": "controller",
  "sub": "71f1e48f-cc58-4ccf-99e1-ca8a3d290c50",
  "typ": "Bearer",
  "azp": "controller",
  "session_state": "73d35069-dfc0-4487-a190-
53da3c3d15a0",
  "acr": "1",
  "realm_access": {
    "roles": [
      "offline_access",
      "admin",
      "grantedTopology",
      "default-roles-sdn",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "authorization": {
    "permissions": [
      {
        "scopes": [
          "post",
          "get",
          "delete",
          "put"
        ],
        "rsid": "463a0819-8c74-4dd4-a1be-d94d4bd1c8be",
        "rsname": "Grants"
      },
      {
        "scopes": [
          "post",
          "get",
          "delete",
          "put"
        ],
        "rsid": "a3ce3aa0-89da-4cd0-8c82-1afde97ae54f",
        "rsname": "Controlador"
      },
      {
        "scopes": [
          "post",
          "get",
          "delete",
          "put"
        ],
        "rsid": "d88fb809-7aa2-4046-b18a-3f42a7e6cf41",
        "rsname": "Users"
      },
      {
        "scopes": [
          "post",
          "get",
          "delete",
          "put"
        ],
        "rsid": "5e73aae9-1563-473c-bc35-55d7ec1a8086",
        "rsname": "Roles"
      },
      {
        "scopes": [
          "post",
          "get",
          "delete",
          "put"
        ],
        "rsid": "4f9d3452-e44d-416f-85e2-84788548ef8a",
        "rsname": "Network-Topology"
      },
      {
        "scopes": [
          "post",
          "get",
          "delete",
          "put"
        ],
        "rsid": "d5852c39-7645-40ec-b7c2-620a5b862793",
        "rsname": "Domains"
      },
      {
        "scopes": [
          "trustHigh"
        ],
        "rsid": "79e0cbb6-4820-41d2-ac3c-e76f0796a2b7",
        "rsname": "Context"
      }
    ]
  },
  "scope": "openid profile email",
  "sid": "73d35069-dfc0-4487-a190-53da3c3d15a0",
  "email_verified": false,
  "appId": "b4a9afb9-e032-5776-9af4-7ebce38532fd",
  "name": "first_name last_name",
  "networkId": "1c4f2663-97ee-59fa-a3e2-1084e85b8a08",
  "preferred_username": "admin@sdn",
  "serviceId": "1ed77836-00ed-543a-9afe-7ae2e428e19b",
  "given_name": "first_name",
  "network_type": "private",
  "family_name": "last_name",
  "deviceId": "4ddba02e-b7f0-5b10-9018-028b9f69c6f8",
  "email": "admin@gmail.com",
  "app_env_type": "e982f17b-cbe0-5724-863b-
708a4f76db21"
}
```


Figure 5.11: Payload Data Of RPT

5.2.2 Relying Party

The RP is the instance that establishes a communication between the controller and Keycloak, mostly through the use of the Keycloak REST API. The use of the Keycloak REST API is to maintain the database regarding user management of the Keycloak synchronized with the database of the controller and its entries of users, and roles. RP is also a crucial component of a OIDC authentication since it is used to authenticate the users in the controller while communicating with the OP Keycloak.

In order to use the Keycloak REST API, the RP must have the credentials, *client id, and secret*, of a client in Keycloak with administrative roles (*"admin-cli"*, recall Section 5.2.1). These credentials and the grant_type *"client_credentials"* are used to request an access token with administrative permissions, which will be used to make user management requests to Keycloak through the Keycloak REST API.

The RP must also have the credentials, *client id, and secret*, of the OIDC client in Keycloak that is going to be used to authenticate the users in the controller, (client *"controller"*), since the RP is also responsible to receive the user's credentials and initialize the authorization code flow with Keycloak to authenticate that user.

Additionally, the RP needs to have the public key of the *RSA* key used for the encryption of the access and RPT tokens, regarding the OIDC client in Keycloak used for authentication

(client *"controller"*), since it needs to decrypt the token to evaluate the content of the authorization data, as well as the expiry date and its authenticity (step explained in detail in Section 5.3.2).

5.2.3 AAA filter

The new AAA filter created, based on Shiro-Based Authorization, will intercept all requests made to the controller. As mentioned previously, it will then proceed to verify if the request contains credentials and redirects the request to the RP for authentication of the user Section 5.3.1, or if it receives a request with a bearer token in the authorization header, will redirect the request to the RP to verify the authenticity of the token and its validity, as well as the user permissions contained in the RPT token, which is explained in Section 5.3.2.

If the token is valid, as well as the permissions, the request made by the user is allowed, and therefore the HTTP operation can proceed. The AAA filter is also responsible to redirect the request to the right module of the controller. So, if a user requests the user REST endpoint in the AAA module, then the AAA filter, will then call the user REST endpoint instances to deal with the request after the verification of the token.

OpenDaylight is divided into modules and therefore can work independently from each module, or altogether. Nevertheless, this means that when working with other OpenDaylight modules, the new AAA filter is not available since it is only present in the AAA module, and therefore the request to the REST endpoints of that module will not be caught by the new AAA filter. This indicates that the filter would only be able to intercept the requests made to the AAA module, as Figure 5.12 (inspired by [64]) suggests, assuming the RESTful API A and northbound plugin A belongs to the AAA module.

So, our implementation suggests and proposes a new OpenDaylight architecture as well, as Figure 5.13 suggests, where there is a AAA filter and a RESTful API common for all modules, meaning that the AAA filter would catch all requests made, whether the request made by a user was to an AAA module REST endpoint or a netconf/restconf module REST endpoint. In other words, We would introduce a central point for authentication, authorization, and policy enforcement in the AAA filter for all operations. This would also improve the security overall of the controller since we only have a single point of entry in the northbound. However, there is also the need to guarantee the existence of the AAA filter and REST API when the modules are working independently.

For the experimental phase of this implementation, we assume a scenario where this new architecture already exists, and therefore as previous Figures of Keycloak configuration have shown, for the REST endpoints regarding the topology management in the netconf and restconf modules, we have already specified roles in Keycloak and controller, as well resources, policies and permissions in the authorization of the OIDC client in Keycloak, which give users permissions to access and request the topology endpoints.

5.3 Steps for Authentication and Authorization

For each step, there will be a diagram with the steps that are performed as well as a description of different operations for a better understanding.

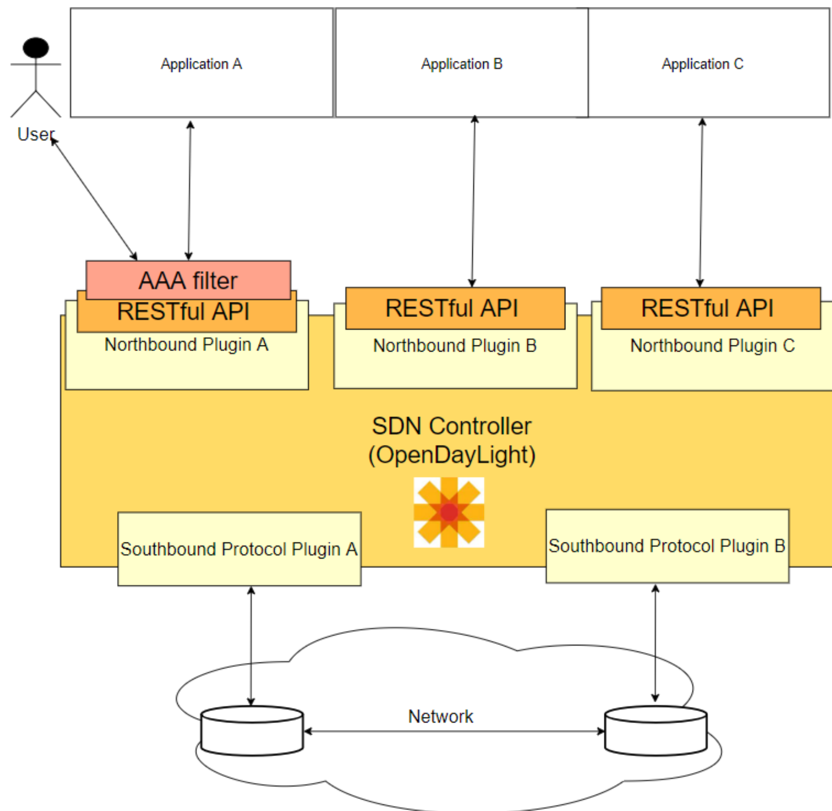


Figure 5.12: Simplified Architecture OpenDaylight Controller

5.3.1 Authentication

As briefly explained in Section 3.5 the AAA filter can receive different types of requests, whether with credentials, with a bearer token, or without any of the two. Upon receiving a request that includes user credentials, it will also verify if the request contains any context information (context information data sent by the user in a JSON format). As Figure 5.14 suggests, if the request contains context information then, both credentials and context information are then passed to the RP, otherwise only the user's credentials are passed. The RP has the responsibility of authenticating the user whose credentials were sent in the request.

If the request contains context information then the RP must first make a request to Keycloak using the Keycloak REST API and fill the user attributes with these values of context received. The context information sent to the user attributes is first converted from a string into a UUID, following the same logic for the experimentation with TeaStore. If the request does not contain context information then the RP must send a request with empty fields to clean previous context information stored in the user attributes from previous authentications.

Following, the RP will finally start the authorization code flow, using the credentials of the user received and the credentials of the OIDC client used for authentication. Originally, in an authorization code flow, following the authentication request, the OP responds with a login form, for the user to fill and proceed with the authentication and begin a valid login session. However, since this authentication is not done through a web context, the RP will receive this login form response and will automatically fill the form with the user's credentials received and submit the form, acting as the user. If everything is successful,

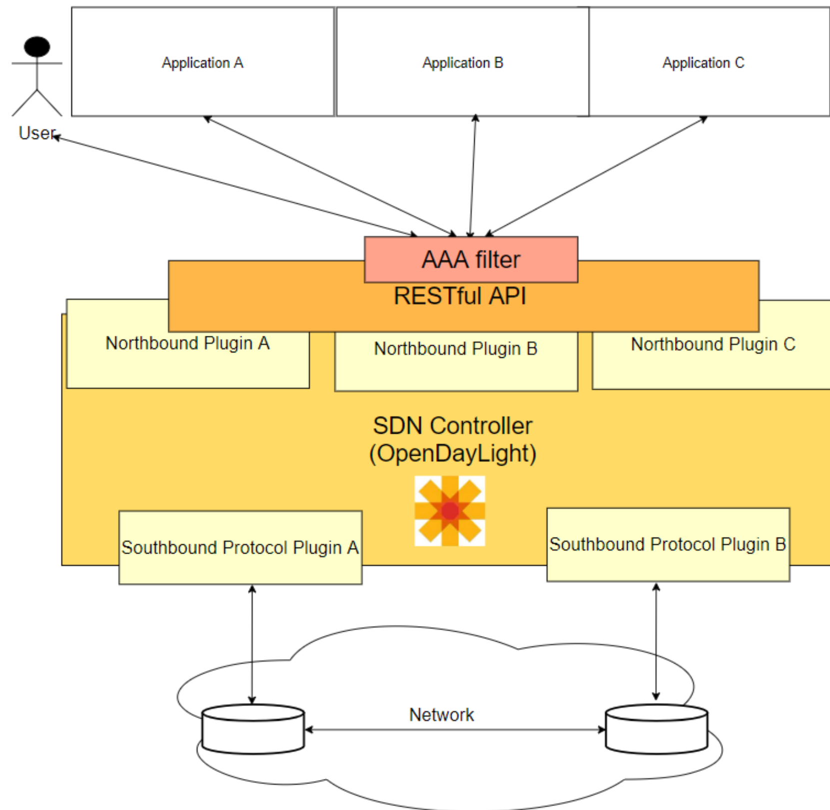


Figure 5.13: Simplified Modified Architecture OpenDaylight Controller

the RP will then receive a code which will exchange through a request to the OP, receiving access, refresh and id token.

However, this access token does not contain the authorization data that is needed to allow a user to make a successful request to the controller, this token needs to be exchanged for an RPT token. To retrieve an RPT token, the RP must send a grant type specified as *"urn:ietf:params:oauth:granttype:uma-ticket"*, an audience and client id, and the access token received as the bearer token in the authorization header. The response of this request will generate a RPT token and a refresh token associated with the RPT token, which is stored in the database of the controller, in the entry of the user authenticated. The id token received from the previous request is also stored in the database. In the end, the RPT token is returned to the user as the token to be used in the authorization header for future requests to the controller.

5.3.2 Verification of RPT token

The first step of the verification of the RPT token is its decryption with the public key mentioned above in the possession of the RP, as per presented in Figure 5.15. Secondly, we verify the username in the decrypted token and we verify if the username contains the *"@sdn"* at the end of the username. If the *"@sdn"* is present, then we know that this user was created in this controller, since all users by default in OpenDaylight when created, have a *"@sdn"* suffix in its userid (this is concerning the only domain existent in the controller called *"sdn"*). In Keycloak, when a user is created through the Keycloak REST API, the username will already include an "@sdn" suffix to match the configuration

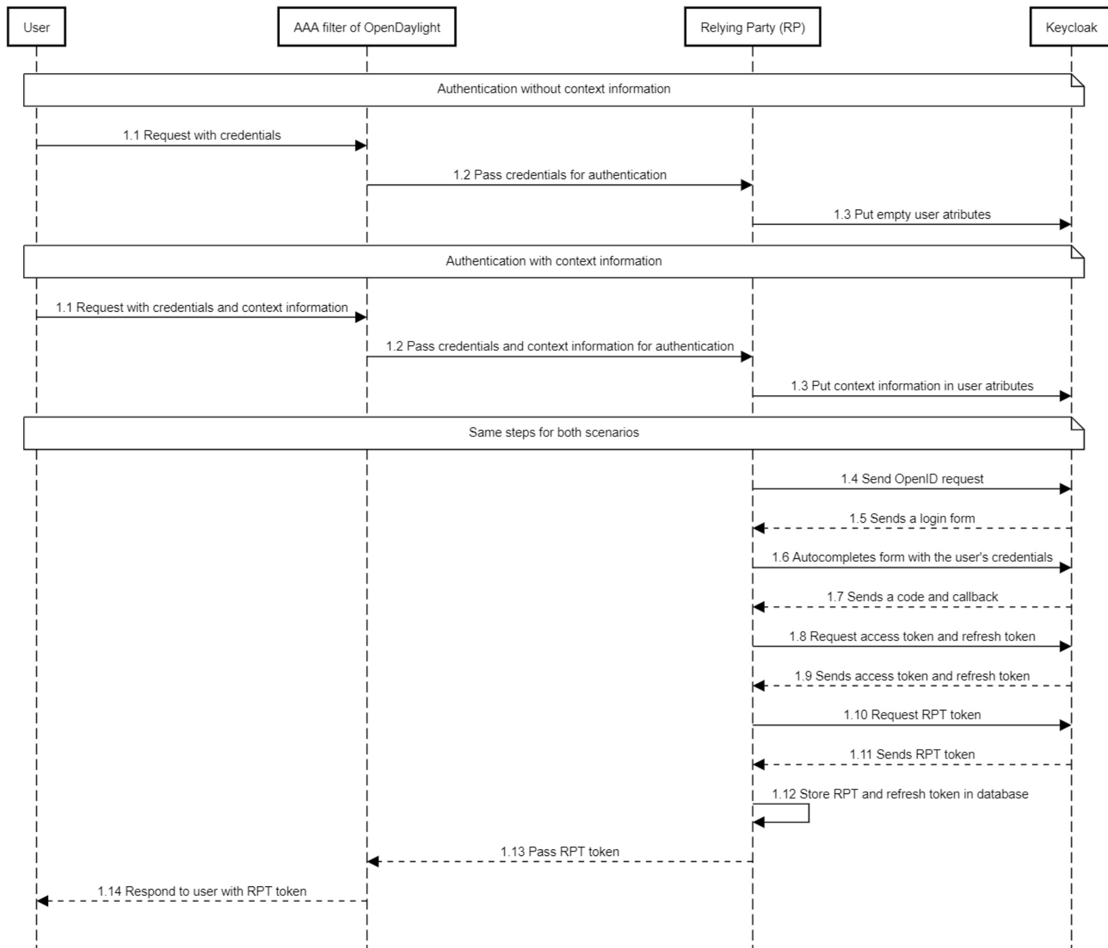


Figure 5.14: Authentication Steps

of the controller and to match the name of the Keycloak realm "sdn". Therefore, while evaluating the decrypted token, if the username does not contain the "@sdn" suffix, this means the user is not registered in this controller, so it does not have permission to make any request to this controller.

Following the username, the expiration time claim in the RPT is retrieved and evaluated. If the remaining lifetime of the token is already less than half, the operation of requesting new tokens for the user will be realized (Section 5.3.3). Otherwise, if the token still has enough lifetime, then we must try to retrieve the context claims in the user attributes of the RPT token. If these context claims are present, we know that the user was authenticated with context information and therefore we need to evaluate it.

For example, as Figure 5.16 present (RPT token is partitioned for better visualization), in the permissions field of the authorization data of the RPT token, there will be a resource with the name *"Context"* and the scope will specify the trust policy that was assigned by Keycloak in the evaluation of the context data. If the policy assigned is *trustLow*, then the user can only make a *GET* request. If the policy assigned is *trustAverage*, then the user can only make *GET and PUT* requests. If the policy assigned is *trustHigh*, then *all types* of requests are allowed. So, additionally, we evaluate the HTTP operation of the request made by the user. If the type of request is allowed, then the verification of the permissions to access the REST endpoint specified in the request is followed. If we find a match between the resources detailed in the RPT token of the user and the endpoint in the request, then the user is allowed to make this request. With a *trustHigh* policy

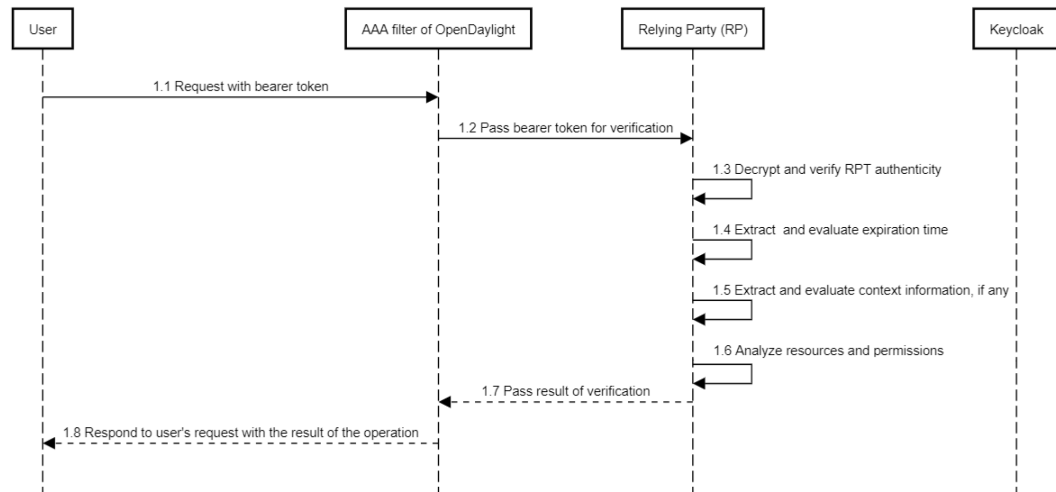


Figure 5.15: Verification Of RPT Token Steps

Figure 5.16: Authorization Data In RPT Token

in the resource *"Context"*, and imagining that a user is making a *POST* request to the grants REST endpoint, the user is allowed to make a *POST* to this endpoint because of the policy assigned, and since it has the *"Grants"* resource specified in the permissions of the RPT token, there is a match between the resource and the request endpoint, so the user's request is allowed.

If a user does not contain context claims in the user attributes of the token, then we must only verify for each resource specified in the authorization data in the RPT token of the user, if a match is found with the resource the user is trying to access with his request. If a match is found, we must also verify if the HTTP operation is in the allowed scopes for that resource. For example, imagining that this RPT token did not include the context information resource, using the same example as the previous paragraph, the RPT token contains the Grants resource, Figure 5.16 , and the scopes of that resource include the *POST* operation. Therefore the user has permission to make a POST to the grants endpoint.

5.3.3 Renewal of RPT token

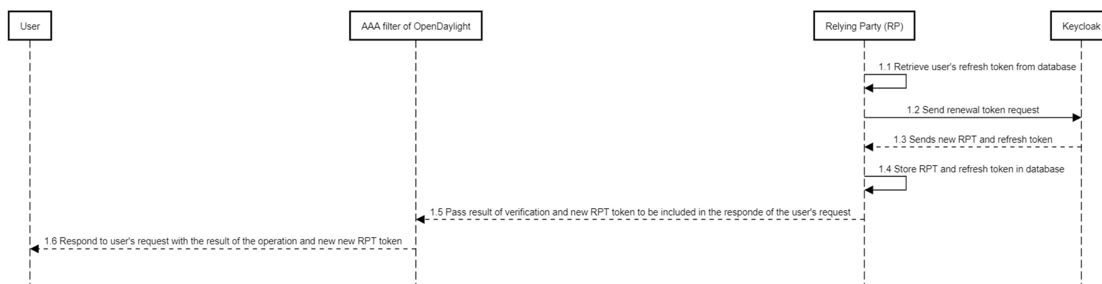


Figure 5.17: Renewal Of RPT Token Steps

During the process of the RPT token verification, if the remaining lifetime of the token is already equal or less than half then a renewal token request will be automatically made to the Keycloak, to maintain the active session of the user (same logic applied for the experimentation with TeaStore, recall Section 4). This value of 50% remaining lifetime before renewal was decided for this implementation, but this value could be 75% or any other justifiable value. For this refresh token operation, Figure 5.17, we need to send the refresh token of the user, the grant type specified of *"refresh_token"*, the client id, and secret of the OIDC client. To retrieve the user's refresh token, the username recovered from the decrypted token during the verification process will be used to query the entry of the user on the controller database. Since the refresh token saved is the one generated along the RPT token, the response of the renewal token request will also be a new RPT and refresh token, instead of an access token and refresh token. These new tokens are saved in the controller database and the new RPT token is returned to the user in addition to the response of the request that the user made. In addition, the limitations for the number of renewal procedures using the same refresh token is dependent on the configurations established on Keycloak. If the option of revocation for refresh is active, each refresh token can only be used once. If this option is not active, the refresh token can be used until its expiration time.

Chapter 6

Setup and Evaluation

This chapter presents the setup prepared for the experimental phase, in order to evaluate the performance and security of OIDC support in OpenDayLight, as well as security gains with the use of context information.

The documented results compare the standard OpenDaylight (i.e. username and password), the support for OIDC and the support of OIDC with context information.

6.1 Experimental Setup

As explained in the use case, Section 3.4, both the context information of the device used to perform operations and levels of authority are relevant for this work.

Therefore, the experimental setup includes four distinct types of test sets prepared, in order to analyze different metrics and different procedures in the controller. The first, **Authentication Process**, gives insight about an OIDC authentication in the controller, as well as the verification of an RPT token received by the user and used in the authentication header in the following requests. The **Refresh/Renewing token process** analyses the process of obtaining a new RPT token, in a token renewal request, within different session times. The third section, **Standard authentication and OIDC authentication in OpenDaylight**, compares an authentication followed by a request, made by an admin, both in the original OpenDaylight controller and in the modified OpenDaylight controller. The last set, **Access Denied**, has the goal to verify the security and functionality of the roles added in OpenDaylight and the use of TCI while making sensible operations in the OpenDaylight.

For all the different tests done, the tokens always have a lifetime of 60 seconds and the test is run 3 times. To test with different scenarios of TCI, it was prepared a file with two different entries for each type of context information. Upon the authentication, one of the entries is arbitrarily chosen according to the respective scenario being tested and sent along with the user credentials. These tests are made with the help of Locust, to automate the user's requests. The number of users for the two first sets of tests was 5, 25, and 50, being 50 already a high value for the use case presented where administrative operations occur in a SDN controller. For the third test set, we only used the *"admin"* user in OpenDaylight and for the final set, we only used 5 users.

OpenDaylight was instrumented to gather performance metrics (i.e. time to execute a certain operation), while the other metrics related to resource usage like CPU utilization,

and amount of memory used, were collected with the user processes scrapper in Prometheus [65]. The specific metrics are presented in the following sections.

6.1.1 Authentication Process

The first set of tests was instrumented to analyze the impact of OIDC in the authentication process. For such analysis, there are different groups of users, 5, 25 and 50 depicted in Table 6.1. Each type of user will do an authentication followed by REST operations in different context scenarios.

Table 6.1: Scenarios Combination For Authentication And Request

Parameter	Values
Number of users	5^{+}, 25^{*}, $50^{\triangle}$
Type of user	admin$^{+,*,\triangle}$, grantedToplogy$^{+,*,\triangle}$, grantedRoles$^{+,*,\triangle}$, grantedGrants$^{+,*,\triangle}$, grantedUsers$^{+,*,\triangle}$, grantedDomains$^{*,\triangle}$, grantedAll$^{*,\triangle}$
Context	No context$^{+,*,\triangle}$, Low context$^{+,*,\triangle}$, Average context$^{+,*,\triangle}$, High context$^{+,*,\triangle}$

Tables 6.2 and 6.3, present the requests that each type of user does in the group of 5, 25, 50 users, for the different scenarios of context (in order of rows: no context, low context, medium context, high context).

Table 6.4, represents the number of instances for each type, in each group of 5, 25, and 50 users.

Before running each group test with 5 users, 25 users, and 50 users, we have to register and create through a *POST* request to the controller the respective 5, 25 and 50 users, in order to be able to authenticate them and perform operations.

Considering, the number of instances per user type, and the different context scenarios, in total, for each 5, 25, and 50 groups, we have 72 requests with no context, 62 requests with low context, 66 requests with medium context, and 66 with high context.

The impact in the authentication process was measured considering the following metrics, Table 6.5:

6.1.2 Refresh/Renewing tokens process

The second set of experiments aims to assess the impact of the renewal process, considering the tests with 5, 25, and 50 users, with different types of users, as well as with sessions lasting 2 and 4 minutes, as depicted in Table 6.6. As we see in Table 6.6, there is no reference to scenarios with different because on this second set we will be only running with no context information. The reason for this decision is based on the fact that this second set was prepared after running the experiments of the first set, Section 6.1.1 and the metric "Get RPT" did not show an impact on whether we used different context information or not (see results in Section 6.2.1).

As mentioned in Section 6.1, each token has a lifetime of 60s and as explained in Section 5.3.3, the renewal process will be triggered if the remaining lifetime of the token is half or less (e.g 30s or less). So, therefore, in a 2 minutes session, there will be 3 renewal occurrences for each user as presented in Figure 6.1.

The same logic is applied to a session of 4 minutes, where there is the first authentication

Table 6.2: Requests For Authentication Test Set

Type user	**5 group operations**	**25 group operations**	**50 group operations**
admin	6x get domains 6x post domains 6x post grants 6x get users	2x get users	1x get users
	6x get domains 6x get roles 6x get grants 6x get users 6x get topology	2x get users 2x get grants	1x get users 1x get grants
	6x get domains 6x get roles 6x get users	2x get users	1x get users
	6x post domains 6x post topology 6x post users	2x post users	1x get users
grantedToplogy	8x get topology	2x get topology	1x get topology
	8x get topology	2x get topology	1x get topology
	8x get topology 8x put topology	2x get topology 2x put topology	1x get topology 1x put topology
	8x delete topology	2x delete topology	1x delete topology
grantedRoles	8x get roles 8x post roles	2x get roles 2x post roles	1x get roles 1x post roles
	8x get roles	2x get roles	1x get roles
	8x get roles	2x get roles	1x get roles
	8x post roles 8x delete roles	2x post roles 2x delete roles	1x post roles 1x delete roles

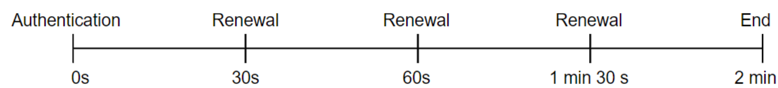


Figure 6.1: Renewal Timeline In A 2 Min Session

Table 6.3: Requests For Authentication Test Set (Continue)

Type user	**5 group operations**	**25 group operations**	**50 group operations**
grantedGrants	8x get grants 8x delete grants	2x get grants 2x delete grants	1x get grants 1x delete grants
	8x get grants	2x get grants	1x get grants
	8x get grants	2x get grants	1x get grants
	8x post grants 8x delete grants	2x post grants 2x delete grants	1x post grants 1x delete grants
grantedUsers	8x post users	2x post users	1x post users
	8x get users	2x get users	1x get users
	8x put users 8x get users	2x post users 2x get users	1x post users 1x get users
	8x delete users	2x delete users	1x delete users
grantedDomains		2x get domains 2x post domains	1x get domains 1x post domains
		2x get domains	1x get domains
		2x get domains	1x get domains
		2x post domains	1x post domains
grantedAll		2x post grants	1x post grants
		2x get domains 2x get roles	1x get domains 1x get roles
	-	2x get roles	1x get roles
	-	2x post topology	1x post topology

and then 7 renewals for each user.

Table 6.7, presents the requests that each type of user does in the group of 5, 25, 50 users. In a session of 2 minutes, these requests are made 4 times (authentication followed by request, and 3 renewals followed by requests each). In a session of 4 minutes, these requests are made 8 times (authentication followed by request, and 7 renewals followed by requests each).

Table 6.4 also represents the number of instances for each type, on each group of 5, 25, and 50 users for the renewals test set. Following the same logic as the previous test set,

Table 6.4: Type Of User Instances

Type user	5 group	25 group	50 group
admin	1	3	6
grantedToplogy	1	4	8
grantedRoles	1	4	8
grantedGrants	1	4	8
grantedUsers	1	4	8
grantedDomains	1	3	6
grantedAll	1	3	6

Table 6.5: Metrics For Authentication And Request

Metric	Description	Unit
Get access token	time it takes to generate an access token from the start of an authorization code flow	ms
Get RPT	time it takes to request a RPT token in Keycloak after receiving an access token	ms
Put user attributes	the time it takes to put the user's attributes in Keycloak, when using context information in authentication request	ms
Verify RPT	the time it takes to verify an RPT received by the controller in a authorization header	ms
Size of RPT	the size of an RPT	bytes
N. Process executing	the number of normal processes executing in user mode	elementary unit
RAM used	the amount of memory RAM used	bytes

Table 6.6: Scenarios Combination For Renewals

Parameter	Values
Number of users	5^{+}, 25^{*}, $50^{\triangle}$
Type of user	admin$^{+,*,\triangle}$, grantedToplogy$^{+,*,\triangle}$, grantedRoles$^{+,*,\triangle}$, grantedGrants$^{+,*,\triangle}$, grantedUsers$^{+,*,\triangle}$, grantedDomains$^{*,\triangle}$, grantedAll$^{*,\triangle}$
Time of session	$120^{+,*,\triangle}$, $240^{+,*,\triangle}$

the 5, 25 and 50 users are created through a *POST* request to the controller before the experimentation.

Considering, the number of instances per user type, and the number of repetitions for each session, in total, we have 288 requests in a 2 minutes session for each 5, 25, and 50 groups and 576 requests in a 4 minutes sessions also for each group.

The impact in the refresh process was measured considering the following metrics, Table 6.8:

6.1.3 Standard authentication and OIDC authentication in OpenDaylight

The third set of tests aims to establish a comparison between the standard authentication (i.e., on username and password) and the OIDC authentication in OpenDaylight. This comparison relies on an admin authenticating and making REST operations on the standard OpenDaylight and the modified OpenDaylight with OIDC augmented with TCI. In this set there are only two HTTP operations, get and post in the three REST endpoints documented in Table 6.9.

Table 6.7: Requests For Renewal Test Set

Type user	5 group operations	25 group operations	50 group operations
admin	6x get domains 6x post domains 6x post grants 6x get users	2x get users	1x get users
grantedToplogy	8x get topology	2x get topology	1x get topology
grantedRoles	8x get roles 8x post roles	2x get roles 2x post roles	1x get roles 1x post roles
grantedGrants	8x get grants 8x delete grants	2x get grants 2x delete grants	1x get grants 1x delete grants
grantedUsers	8x post users	2x post users	1x post users
grantedDomains	-	2x get domains 2x post domains	1x get domains 1x post domains
grantedAll	-	2x post grants	1x post grants

Table 6.8: Metrics For Renewal

Metric	Description	Unit
Get new RPT	time taken to generate a new RPT in a renewal token request to Keycloak	ms
N. Process executing	the number of normal processes executing in user mode	elementary unit
RAM used	the amount of memory RAM used	bytes

Table 6.9: Requests On Standard And Modified OpenDaylight Controller

Parameter	Values
Endpoint request	Domains^{+}, Users*, Grants$^{\triangle}$
Type of user	admin$^{+,*,\triangle}$
HTTP operation	get$^{+,*,\triangle}$, post$^{+,*,\triangle}$

In the third set, the following metrics were considered, Table 6.10:

Table 6.10: Metrics For Standard And OIDC Opendaylight Authentication

Metric	Description	Unit
Authentication:	the time it takes to authenticate a user in OpenDaylight controller	ms
REST request	time taken to complete REST operations in Opendaylight	ms

6.1.4 Access denied

For the last set of tests, the goal was to analyze the security and functionality of the roles added in OpenDaylight and the use of TCI while making sensible operations. So, there are two cases we want to analyze in this section. The first case is when a user does not have the right permissions to access a specific restricted endpoint in OpenDaylight. And the second one is when a user has the right permissions to make a sensible operation on the controller however the context of the authentication does not portray high trust.

For such analysis, the group of 5 users is sufficient to analyze these cases and are presented in Table 6.11. Each type of user will do an authentication followed by REST operations in different context scenarios.

Table 6.11: Scenarios Combination For Access Denied Test Set

Parameter	**Values**
Number of users	5^+
Type of user	admin$^+$, grantedToplogy$^+$, grantedRoles$^+$, grantedGrants$^+$, grantedUsers$^+$, grantedDomains$^+$, grantedAll$^+$
Context	No context$^+$, Low context$^+$, Average context$^+$, High context$^+$

Table 6.12 presents the requests that each type of user does in the group of 5 for the different scenarios of context (in order of rows: no context, low context, medium context, high context). For the scenarios with no context and high context, the focus is on the security portrayed by the roles and permissions, since in these scenarios all types of operations are allowed, and for the scenarios of low context and average context, the focus is on the context information for more sensible operations (For example, a grantedUsers in a low context scenario tries to make a *POST* request to the users REST endpoint).

Before running each group test with 5 users, we have to register and create through a POST request to the controller the respective 5 users, to be able to authenticate them and perform operations.

Considering, the number of instances per user type (1 for each), and the different context scenarios, in total, 20 requests are being made to the controller.

The only measurements made for this set are the receiving of success or unauthorized response.

6.2 Evaluation Results

The results obtained from the experimental setup are presented per test set in the following sections. For all boxplots presented, 'x' represents the mean.

6.2.1 Authentication Process

The authentication process performance is assessed in terms of getting an access token, "Get access token", and obtaining a RPT token, "Get RPT", which are the result of successful authentication, and sending context information to an OP for later evaluation, "Put user attributes". Following the successful authentication, the performance of verifying an RPT, "Verify RPT", and the size of that token, "Size of RPT", is also assessed.

Table 6.12: Requests For Access Denied Test Set

Type user	5 group operations
grantedToplogy	1x get roles 1x put topology 1x post topology 1x get roles
grantedRoles	1x delete topology 1x delete roles 1x post roles 1x delete topology
grantedGrants	1x get domains 1x post grants 1x delete grants 1x get domains
grantedUser	1x post grants 1x put user 1x post user 1x post grants
grantedDomain	1x put user 1x delete domains 1x post domains 1x put user

Figure 6.2 presents the time in ms of obtaining an access token, since the beginning of an authorization code flow in the OpenDaylight controller. The introduction of context has a low impact on the authentication process since the average time for the majority of the cases is below 175ms. With the number of users increasing the time tends to decrease, showing some effect of Keycloak caching. Regarding the type of user, I would say that the

difference is low with a tendency to have higher variation with admin, grantedRoles, and grantedDomains.

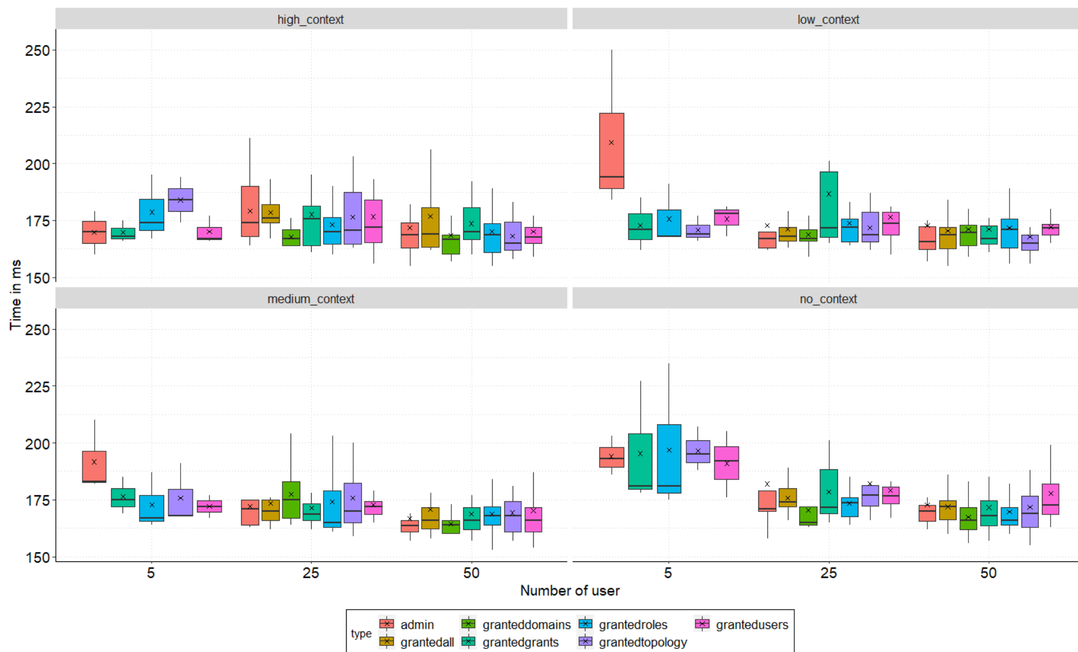


Figure 6.2: Get Access Token

The time, in ms, needed to get an RPT token is represented in the Figure 6.3. The average time obtained for the majority of the cases is between 13-15ms. This value is extremely low compared to the previous figure, where we obtain an access token, however, the previous metric measures the complete OIDC flow, whereas the "Get RPT" metric only measures the exchange of an access token for an RPT token. Regarding the different user groups, the differences are low, where the caching factor of Keycloak is not as noticeable as in the previous Figure 6.2. Compared to the 'no context' scenario, the scenarios with context information also result in a similar performance introducing almost no impact. Regarding the type of user, the difference is also low with a tendency to have higher variation with admin, grantedTopology, and grantedUsers.

Figure 6.4, represents the time needed in ms, to send the context information from the RP to Keycloak, to update the user's attributes. When comparing the different scenarios, even updating the user's attributes with diverse context information, no impact is shown. For most cases, the average presented is around 5ms. Regarding the type of user, the difference is extremely low with a tendency to have higher variation with grantedGrants, and grantedUsers.

The time in ms, it takes to verify a RPT token received by the controller is presented in Figure 6.5. For almost all cases, the average of verifying a token is between 1.5-2.5ms. This process of verification time of an RPT token is almost similar in the tests no matter the different configurations for the type of users. During this process of verification, when context information is present, there is an additional step to verify the policy assigned in the RPT token, however, it does not show impact since the performance for all types of context or no context is similar.

The size of an RPT token is represented in Figure 6.6. Even with the increase in users, the values obtained are very identical. Regarding the type, the admin and grantedAll tokens are bigger, because they have more authorization data. For the different scenarios with

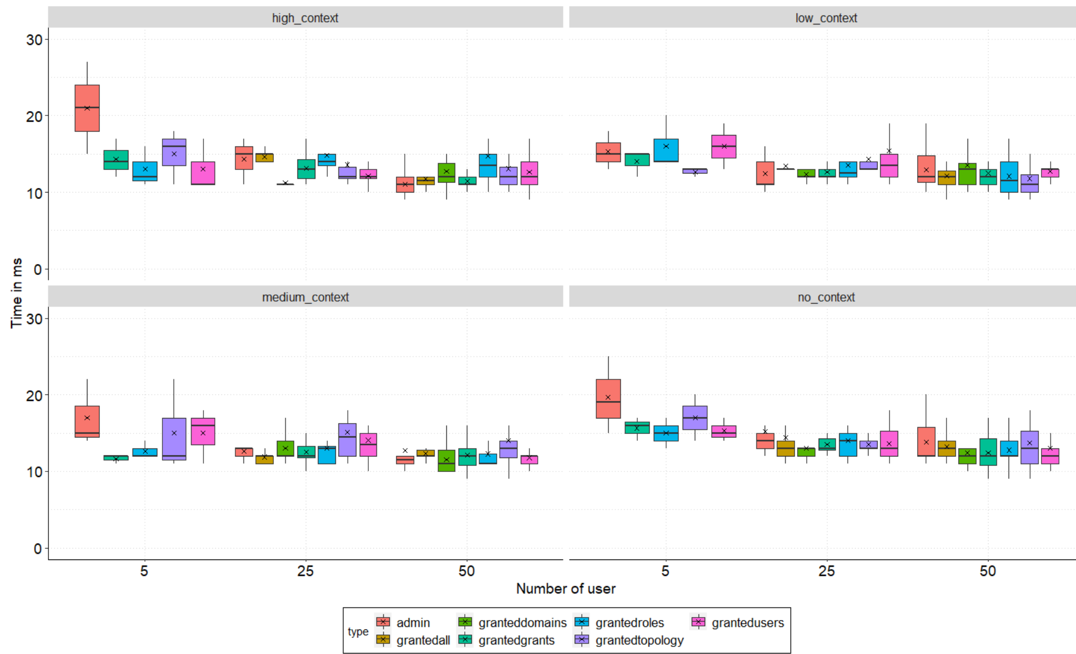


Figure 6.3: Get RPT

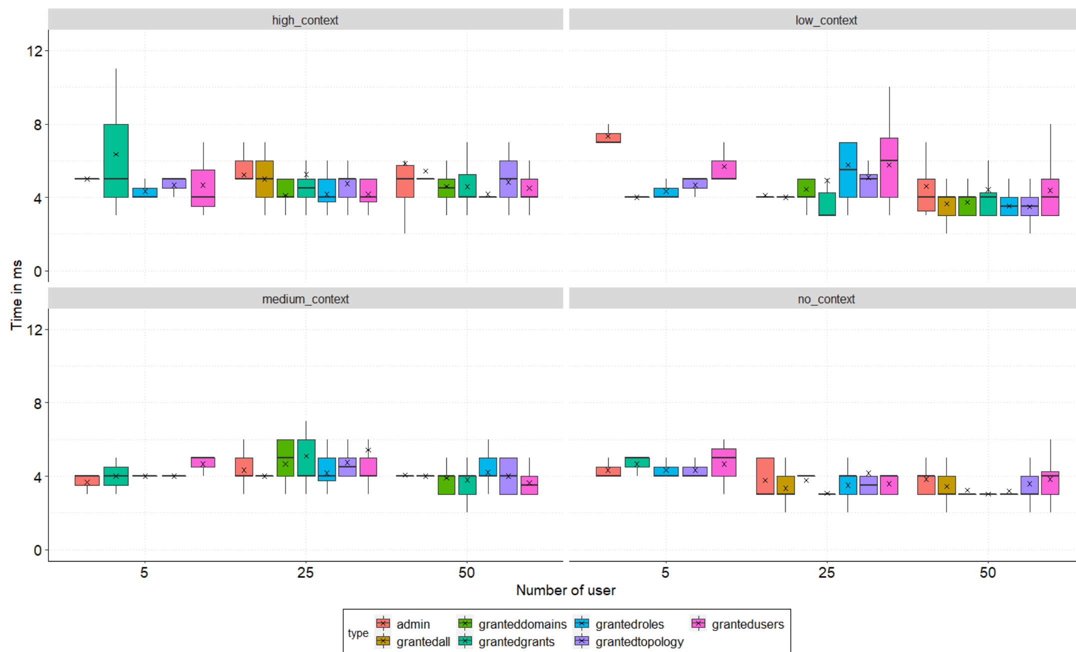


Figure 6.4: Put User Atributes

context, the scenarios with context information will directly translate into more data in the token than the authentications done without context information.

For the 5 group set, the *"admin"* used for testing was the *"admin"* previously created in Keycloak and contains information such as first name and last name as mentioned in Section 5.2.1. For the group of 25,50 as explained in the experimental setup, the users are created through requests to the controller and the only information given to Keycloak is the username and password. Since the *"admin"* instances of 25 and 50 do not contain that extra information as the *"admin"* used in the groups of 5, the size of their tokens is just a little smaller compared to the latter.

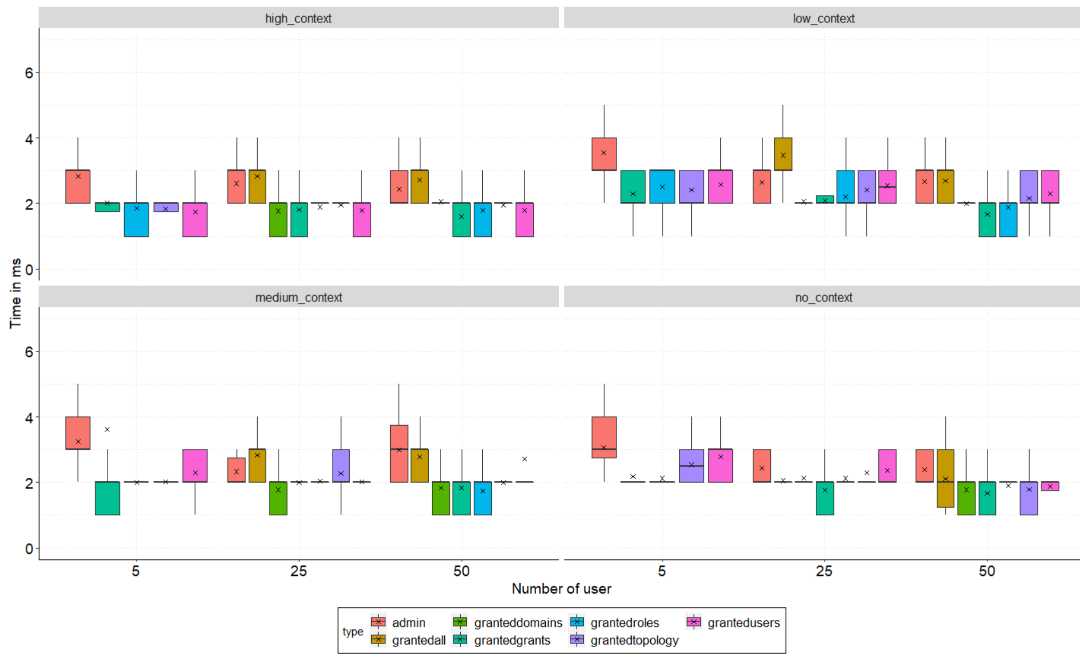


Figure 6.5: Verify RPT

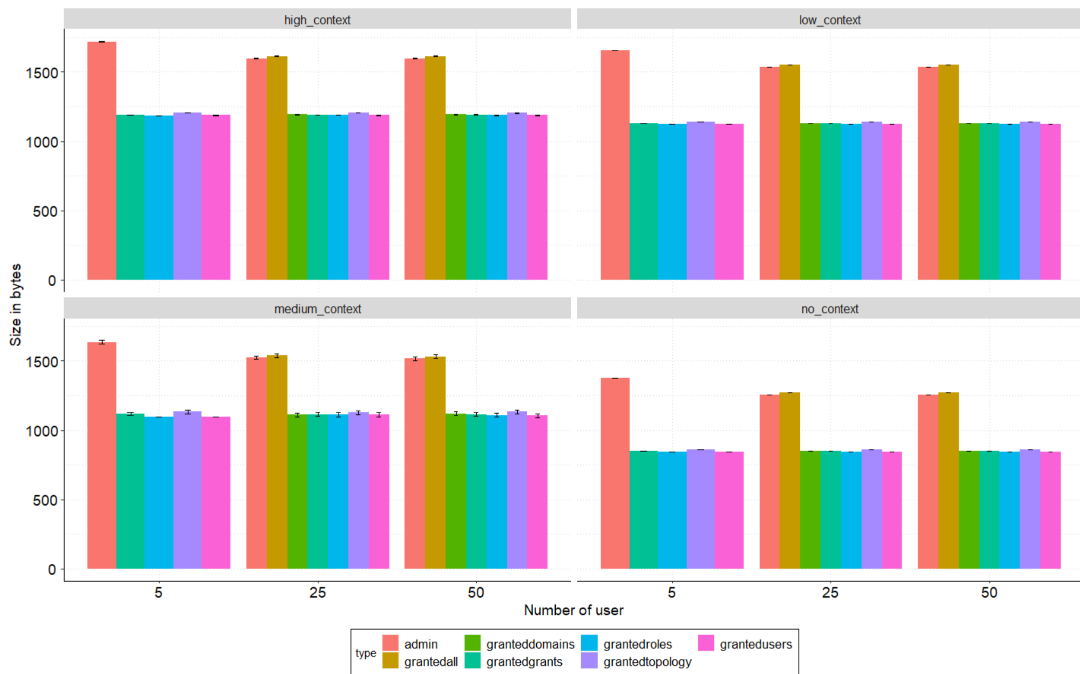


Figure 6.6: Size Of RPT

The next Figure 6.7 presents the RAM used. The processes that were executing in user mode during the different tests are extremely similar to all cases, therefore there is no need for a graph. The RAM consumed, presents a slight increase with 25 and 50 respectively, when compared with the RAM consumed for the tests with 5 users, which is expected, since 25 and 50 brings a bigger load to the system.

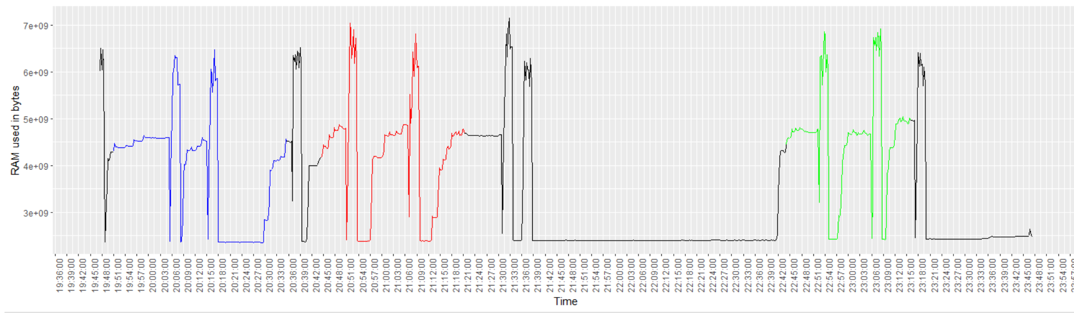


Figure 6.7: RAM Used (5 Group; 25 Group, 50 Group)

6.2.2 Refresh/Renewing tokens process

The Refresh/Renewing tokens process performance is assessed in terms of getting a new RPT token, "Get new RPT".

Figure 6.8 represents the time in ms needed to obtain a new RPT token in a renewal token request to Keycloak, in a session of 2 minutes and 4 minutes respectively. The refresh time is almost similar for the sessions with 2 and 4 minutes, having an average for almost all cases between 10 and 12.5 ms. The results with 4 minutes are more stable (in a 4 minute session there are more renewals so we can see some of the effects of caching in Keycloak) and the admin type user overall tends to present higher values for the refresh.

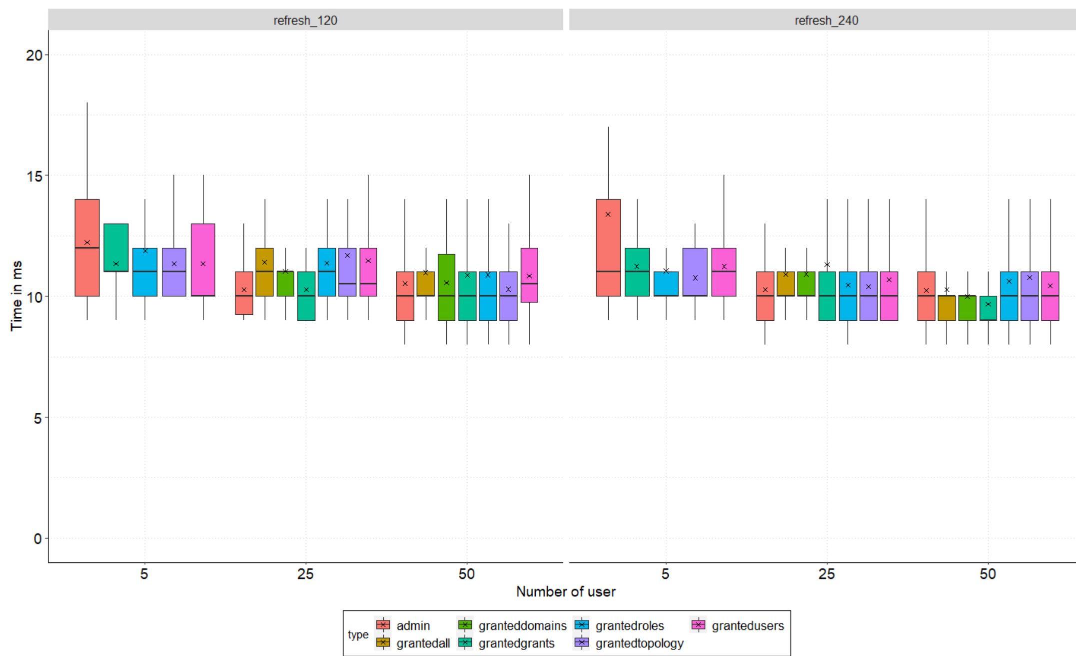


Figure 6.8: Get New RPT

Figure 6.9 presents the RAM used. The processes that were executing in user mode during the different tests are extremely similar to all cases also in the refresh scenarios, therefore there is no need for a graph. The RAM consumed, presents higher peaks with 25 and 50, when compared with the RAM consumed for the tests with 5 users, which is expected, with a bigger load into the system.

Figure 6.9: RAM Used (5 Group; 25 Group, 50 Group)

6.2.3 Standard authentication and OIDC authentication in OpenDaylight

The standard authentication and OIDC authentication in Opendaylight performance is assessed in terms of comparing diverse REST operations plus the time of the authentication.

For the tests with the default admin in the OpenDaylight controller default and the one modified for this work, Figure 6.10 shows the cost of some REST operations and authentication from the admin type of users. **Even though the value of the OIDC authentication of the "admin" is not highlighted in the figure, the value we collected was 411 ms**. The most noticeable factor here is the values of the operations with the admin default in the modified OpenDaylight controller being much higher than the values in the standard OpenDaylight controller. This increase is expected, even though, the values presented are in ms, a complete authentication goes through a lot of steps until a user received a RPT token. However, to increase the security in the OpenDaylight controller, a small cost in performance must be given. Regarding the HTTP operation cost, get operations to have a slightly lower cost than post operations as expected, and the REST operations to the user endpoint are the ones with bigger values, especially the post to the user endpoint. This is expected because, besides the registration of a user in the controller, it also reads the role in the JSON data received in the HTTP request, and it also creates a grant for that user with that role. In the OpenDaylight controller modified, there are two additional operations which are the registration of the user in Keycloak, and the association of a role to that user as well.

With the removal of the OIDC authentication cost in the OpenDaylight controller modified, Figure 6.11, shows a closure look at the time of the rest operations. All values obtained previously with the admin in the controller modified are almost 10 times lower in this new Figure, except the post request for the user endpoint, as already explained, there are multiple operations within that request. So, the cost of these operations themselves is not high, but the authentication cost to retrieve a RPT token to use on the REST operation is the only downfall, in order to have better security.

6.2.4 Access denied

The Access denied performance is assessed in terms of obtaining a successful or an unauthorized response.

As expected from our framework, all the requests return a response with the following message: **Access not allowed or token not valid, please authenticate again**

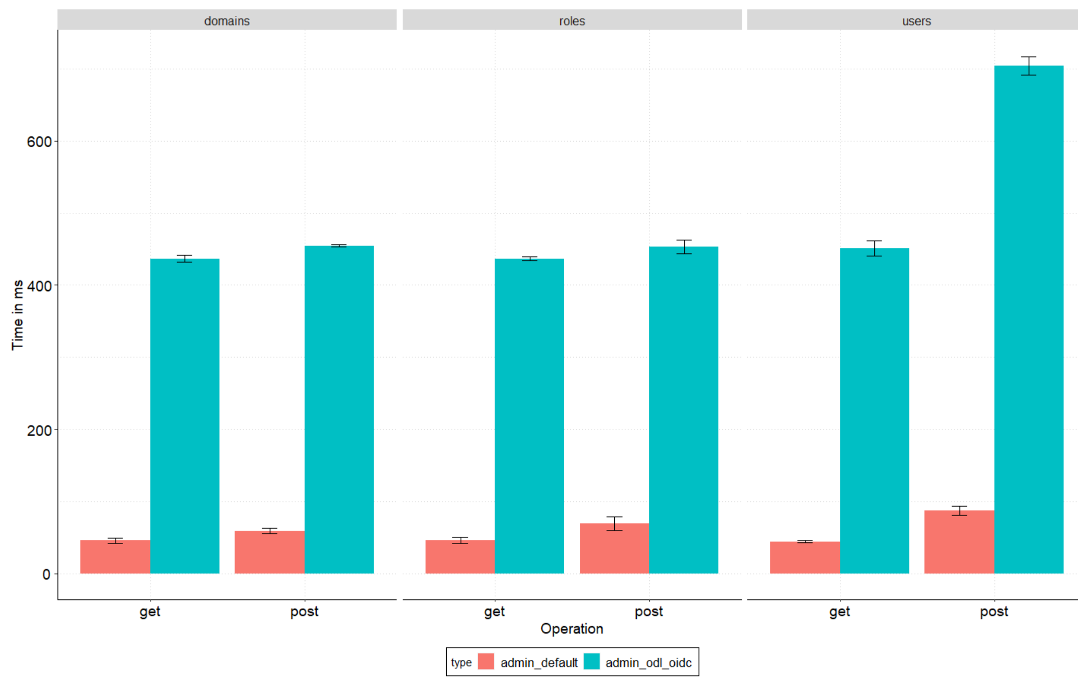


Figure 6.10: Rest Operations With Authentication Time In OpenDaylight Standard And OpenDaylight With OIDC

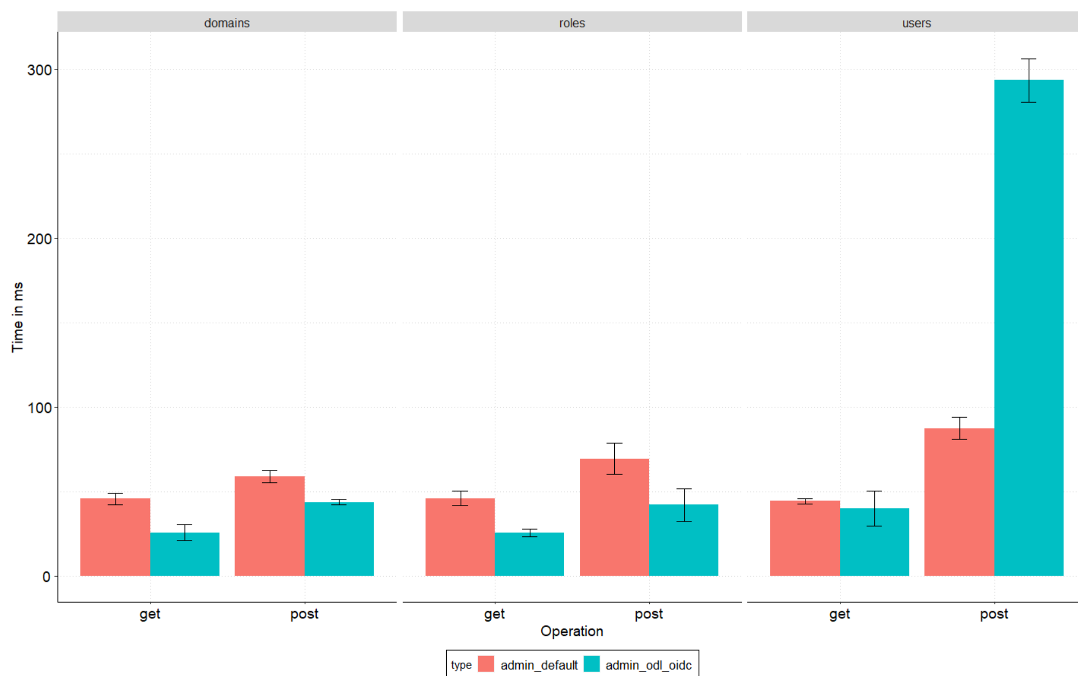


Figure 6.11: Rest Operations With Authentication In Standard OpenDaylight And Only Rest Operations In Modified OpenDaylight

6.3 Enhanced security with OIDC

This section evaluates the impact on the security of adding OIDC support in an OpenDaylight controller as well as the support for OIDC with an extension for context information.

6.3.1 Security with OIDC without context information

From the state of art analyzed in the previous chapter, OIDC was the authentication mechanism that presents more security advantages when compared to others such as OAuth 2.0 and basic authentication schemes. In this scenario, the replacement of the basic authentication mechanism in OpenDaylight with OIDC, using Keycloak improves the security of the controller. The security gains rely on the generation of a RPT token with all the authorization data of a user, which, at any point can be revoked.

Additionally, the results presented in the previous section, showed that this increase in security would have a partial impact on the time of the authentication when compared to the standard mechanism in OpenDayligth. However, the advantages of having the user's permissions on a simple RPT token, helps the following request be more secure and in general fast, since, for every REST operation, there is no need for authentication, since a token has a specified lifetime.

Broders et al, in [66] proposed the use of modeling techniques, including Attack trees, to compare the security of two authentication mechanisms, where this approach showed a detailed analysis of threat coverage of authentication mechanisms as well as their complexity. Following the approach proposed, we also constructed some Attack Trees [1] with the goal of **"Authenticated user makes unwanted sensible operations in OpenDaylight"**, with a few simple exploits presented (i.e, sensible operations could be the creation of a grant to associate a user with the role *"admin"*, remove devices in a topology).

Therefore, Figure 6.12 represents the proposed Attack Tree [AT-01] in the standard OpenDaylight. This small tree shows that to achieve the goal of this attack, they need to send fake/stolen credentials to be able to authenticate or send personal credentials that are authorized, but they are abusing their privileges or have more privileges than they should, and make a request to a restricted endpoint.

Figure 6.13 represents the proposed Attack Tree [AT-02] in OpenDaylight with OIDC and we can see the increase of the tree with the increase of the complexity of the attack. To be able to do an operation in OpenDaylight with OIDC, first, we need to authenticate, and following we send the RPT token received in the authorization header of the wanted request. Therefore, there are three ways to be able to have an RPT token but all of them are complex. First, they need to send the fake/stolen credentials or send personal credentials, while abusing their privileges and obtaining a valid RPT token, which can be revoked at any time and has a lifetime duration. Second, they could try to forge an RPT token with a valid signature, but for that, they need the private key of the signatures which is stored securely in the Keycloak. The last option, is they steal an RPT token through the use of different exploits.

6.3.2 Security with OIDC and trust context information

As explained in the Research Methodology the concept of TCI and as presented in the use case, TCI can bring a more sense of trust to the user as well to the framework in use. Sensible operations to the controller will only be possible if the context information of the user's device is evaluated as of high trust. The results obtained in the previous section showed that even when working with different types of context information, the performance in general of different operations is very similar, not causing a significant impact. Therefore, as the controller is a critical and central point in a network, besides

[1] The modeling of all attacks was not a part of the scope of this thesis

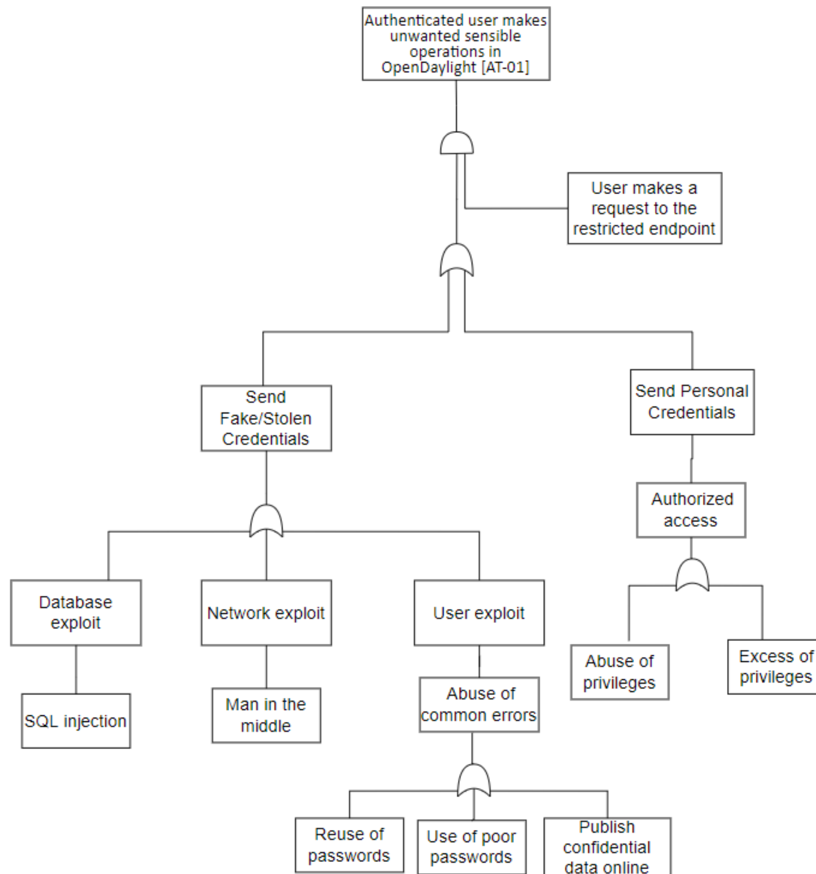


Figure 6.12: [AT-01] - Attack Tree In Standard OpenDaylight

having a secure authentication method, such as OIDC, restricting its access based on the context of the device used for authentication brings an extra layer of security.

The Attack Tree in Figure 6.14 represents the proposed Attack Tree [AT-03] in OpenDaylight with OIDC and support for trust context information, and the exponential growth of complexity is evident. Completing the previous Attack Tree presented, 6.13, if the attacker uses a forged or stolen RPT token, this token must include context information of high trust to be able to make a successful sensible operation in the Controller. If the user is trying to use the fake/stolen credentials or send personal credentials, they also need to send context information of high trust in the authentication request. And if the user is not authorized to do sensible operations that need to be made in a context of high trust, then the attacker only has two ways to find the right context information. The first is by trial and error, and the second one is to gain access to Keycloak user management interface or database and use the context information stored in the Javascript Policy of High Trust. And even after, discovering the context information, since they are stored as UUID, the attacker needs to decode the UUID, in order to send the context information in String format in the authentication request.

6.3.3 Comparison of OpenDaylight security integrations

Table 6.13, shows the security of each integration in OpenDaylight, by analyzing the complexity of each Attack Tree, while comparing the number of OR's and AND's in a tree. The increase in the number of OR's indicates the growth of a tree, and the increase in the

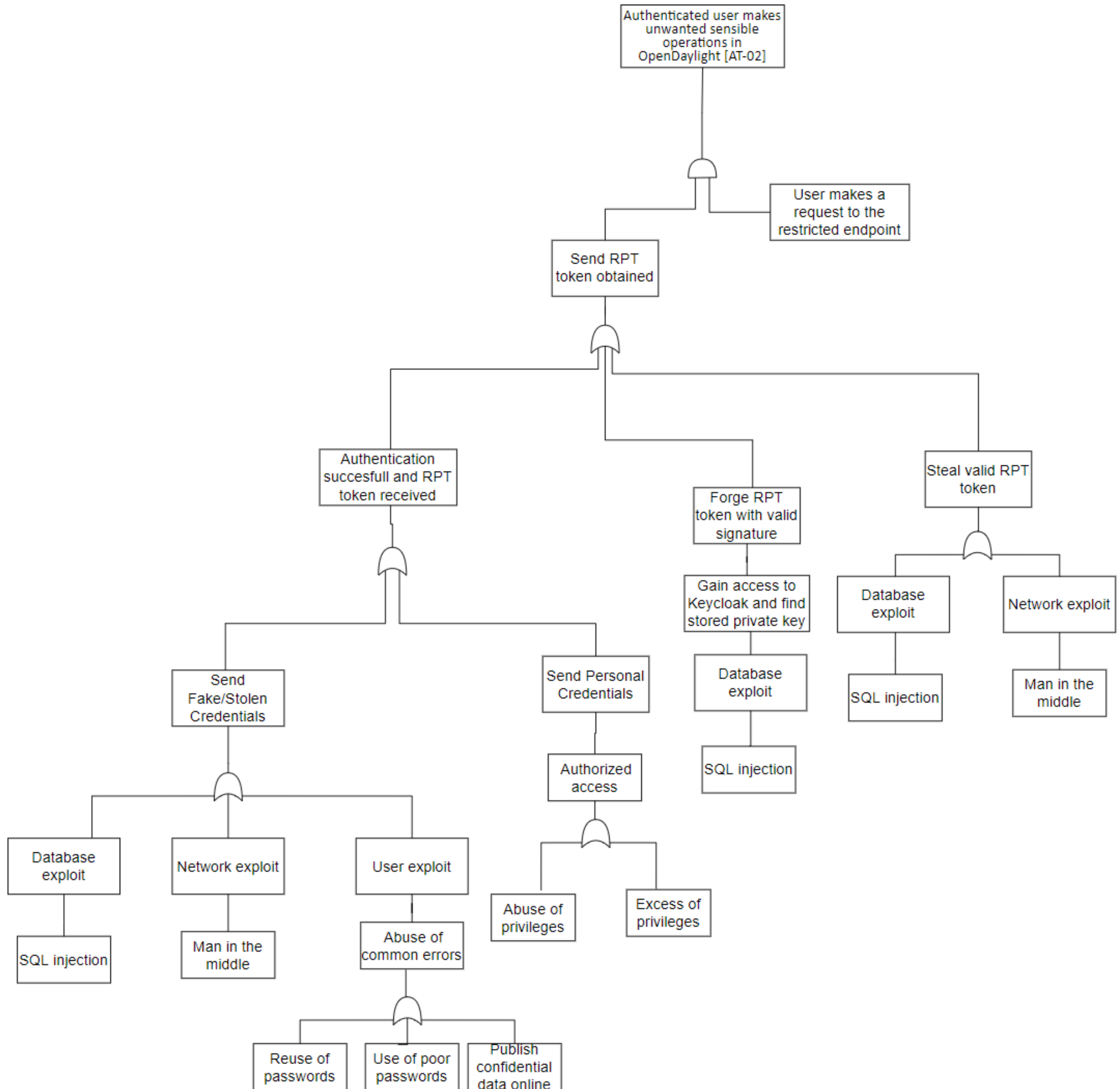


Figure 6.13: [AT-02] - Attack Tree In OIDC OpenDaylight

number of AND's represents the increase in the complexity of the tree since more exploits or attacks need to be combined to achieve the final goal. As proposed by this work, the security of OpenDaylight increases with each security element introduced, where our final integration, shows to be a more difficult target to have a successful attack.

Table 6.13: Attack Tree Comparison - OpenDaylight Security Integrations

Integration	**Nº of OR's**	**Nº of AND's**
Standard Opendaylight	4	1
Opendaylight with OIDC	6	1
Opendaylight with OIDC and context information support	13	5

Figure 6.14: [AT-03] - Attack Tree In OIDC OpenDaylight With Support For TCI

This page is intentionally left blank.

Chapter 7

Conclusion

Nowadays, SDN is even more crucial in network management with the increase of its usage in cloud computing, as well as being one of the key technologies in 5G networks. However, the basic authentication it possesses is vulnerable to different kinds of exploits. Therefore, the main goal of this work is to design and evaluate a framework, which incorporates robust authentication and accounting mechanisms in SDN controllers.

To accomplish the proposed objective, we started by analyzing different authentication mechanisms, where the chosen one was OIDC which uses bearer tokens. The search for a SDN controller to make the integration with OIDC was also extensive. OpenDayLight was the SDN controller chosen based on the documentation available and the superior support for authentication modules.

It also introduced the idea of trust, with the use of a user's device context information to regulate different levels of trust towards the authentication. Each level of trust will correspond to a policy set during the verification of the context information received by the OP during the authentication request. Each policy will result in the presence of different user information inside an ID token, where the higher trust values the more information can be included.

The results obtained evaluated the impact of introducing fields in the claims of OIDC regarding a user's device context information, which indicated a small cost. The relevant values of cost in OIDC with TCI, compared with a standard OIDC implementation, were mainly found in the time taken to verify the different values of context in OP since a standard version of OIDC does not have values of context that must be verified. Measures in TeaStore, with a different number of users, showed some variance in terms of performance, justified by the interaction of different services during an operation.

In the second semester, we started developing the framework proposed, based on the architecture designed and introduced in the first semester, with the OpenDaylight controller, AAA filter, RP, and Keycloak. Then, we introduced the ideas of roles and grants on the controller and how to define a more precise authorization to the various REST endpoints of the controller. We also discussed the setups on Keycloak, the adjustments made to the controller's AAA filter, the addition of the RP component, the processes for authentication, where to redirect each request that is received on the controller, and how to validate a RPT token, while proposing additionally, a new architecture for the OpenDaylight controller.

We created an experimental procedure to test various authentication-related parameters, such as the verification of a token sent by an authenticated user and retrieve metrics related to a token renewal, before comparing the behavior of different REST requests for the

same user, *"admin"*, in both the standard OpenDaylight and the modified OpenDaylight of this framework. We also analyzed the security and functionality of the roles added in OpenDaylight and the use of TCI to make sensible operations (e.g. create grant to associate a user with the role of *"admin"*, remove devices in a topology), to the controller. In this experimental procedure, we varied the number of users, the type of users as well the context of the authentication.

From the results obtained, we observed that even with a different number of users, user type, or context information, the values obtained were very similar in most cases. No major differences were found for the metrics related to authentication and the renewal procedure, except for the "Size of the token" metric, which is expected to have differences since each user type with different context information, will result in more or fewer authorization data stored in a RPT. The test set related to the access denied showed that the roles added prevented unauthorized requests and the use of context information prevented sensible operations to be realized under scenarios where the trust in the context information is low or average.

The major differences were found using the same user *"admin"* and comparing the timings acquired from the controller of these two frameworks, it can be shown that the timing of authentication is approximately 8 times longer and some requests to the controller that also cause modifications in Keycloak are almost 3 times longer.

Although there is a cost associated with increasing the level of security in the OpenDaylight controller, which is less than a second overall, the results obtained were expected and this increase is justified as it gives bigger security by having the advantages of a OIDC authentication and having the trust policies established in regards to the context information of the authentication.

This security gain was proven following a methodology of using attack trees to compare different authentication mechanisms while analyzing the threat scope and complexity of the tree. The attack trees designed showed that the security of OpenDaylight increased with the introduced TCI by showing an increase in the required complexity of the attacks (i.e. an increase of AND's).

7.1 Future work

Future work includes the restructuration of the OpenDaylight architecture as proposed, to have a central point in the controller, the AAA filter, for all types of operations, where the authentication, authorization, policy enforcement, and verification of the token can occur, before a REST request is authorized.

Additionally, this work can lead to the removal of the database of the controller regarding the AAA services, since all the user management is majority realized by Keycloak. There would only be the need to have an entry for each userid, with its respective refresh token, to maintain the user's session valid in longer sessions, by renewing the RPT token.

All the requests performed in OpenDayligth were HTTP requests and not HTTPS requests since the controller didn't have support for it. A new security measure to add to the controller is the addition of support to HTTPS requests, which would give another layer of security by using encryption.

At last, the automation of the detection of the context information is something to explore so we can guarantee its authenticity. This automation and the use of the TCI in OIDC

could lead up to be useful for future projects since nowadays there are more examples of users authenticating in the same application under different context scenarios.

References

[1] Abdallah Moubayed, Ahmed Refaey, and Abdallah Shami. Software-defined perimeter (sdp): State of the art secure solution for modern networks. *IEEE Network*, 33(5):226–233, 2019.

[2] Shi Dong, Khushnood Abbas, and Raj Jain. A survey on distributed denial of service (ddos) attacks in sdn and cloud computing environments. *IEEE Access*, 7:80813–80828, 2019.

[3] Ahmed Sallam, Ahmed Refaey, and Abdallah Shami. On the security of sdn: A completed secure and scalable framework using the software-defined perimeter. *IEEE Access*, 7:146577–146587, 2019.

[4] P. Goransson, C. Black, and T. Culver. *Software Defined Networks: A Comprehensive Approach*. Elsevier Science, 2016.

[5] Openid connect. `https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth`. Accessed: 2021-09-20.

[6] Oauth 2.0 rfc. `https://datatracker.ietf.org/doc/html/rfc6749`. Accessed: 2021-09-20.

[7] Open network operating system. `https://www.onap.org/`. Accessed: 2021-09-30.

[8] Opendaylight. `https://www.opendaylight.org/`. Accessed: 2021-09-27.

[9] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[10] Teastore. `https://github.com/DescartesResearch/TeaStore`. Accessed: 2021-10-17.

[11] Oauth 2.0 section 1. `https://datatracker.ietf.org/doc/html/rfc6749#section-1.1`. Accessed: 2021-09-20.

[12] Oauth 2.0 section 1.2. `https://datatracker.ietf.org/doc/html/rfc6749#section-1.2`. Accessed: 2021-09-20.

[13] Yvonne. Wilson and Abhishek Hingnikar. *Solving Identity Management in Modern Applications Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0*. 1st ed. 2019. edition, 2019.

[14] Oauth 2.0 section 4.1. `https://datatracker.ietf.org/doc/html/rfc6749#section-4.1`. Accessed: 2021-09-20.

[15] Spring boot 2.1: Outstanding oidc, oauth 2.0, and reactive api support. https://developer.okta.com/blog/2018/11/26/spring-boot-2-dot-1-oidc-oauth2-reactive-api. Accessed: 2021-09-25.

[16] Openid authorization code flow. https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth. Accessed: 2021-09-25.

[17] Openid claims. https://openid.net/specs/openid-connect-core-1_0.html#Claims. Accessed: 2021-10-10.

[18] Siriwardena P. *OpenID Connect in Action*. Manning, 2021.

[19] Google-sign-in. https://www.wappalyzer.com/technologies/social-logins/google-sign-in. Accessed: 2021-10-20.

[20] Etsi ts 123 501 version 16.6.0. - system architecture for the 5g system (5gs). https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf.

[21] A comparative introduction to 4g and 5g authentication. https://www.cablelabs.com/insights/a-comparative-introduction-to-4g-and-5g-authentication. Accessed: 2021-10-3.

[22] Yustus Eko Oktian, SangGon Lee, HoonJae Lee, and JunHuy Lam. Secure your northbound sdn api. In *2015 Seventh International Conference on Ubiquitous and Future Networks*, pages 919–920, 2015.

[23] Yustus Eko Oktian, Sang-Gon Lee, and JunHuy Lam. Oauthkeeper: An authorization framework for software defined network. *Journal of Network and Systems Management*, 26(1):147–168, Jan 2018.

[24] Erik Berdonces Bonelo. OpenID Connect Client Registration API for Federated Cloud Platforms. Master's thesis, Aalto University. School of Science, 2017.

[25] Lauritz Holtmann. Single Sign-On Security:Security Analysis of real-life OpenID Connect Implementations. Master's thesis, RUHR-UNIVERSITAT BOCHUM, 2020.

[26] Nitin Naik and Paul Jenkins. Securing digital identities in the cloud by selecting an apposite federated identity management from saml, oauth and openid connect. In *2017 11th International Conference on Research Challenges in Information Science (RCIS)*, pages 163–174, 2017.

[27] Jim Basney, Phuong Cao, and Terry Fleury. Investigating root causes of authentication failures using a saml and oidc observatory. In *2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys)*, pages 119–126, 2020.

[28] Alejandro Pérez Méndez, Gabriel López Millán, Rafael Marín López, David W. Chadwick, and Ioram Schechtman Sette. Integrating an aaa-based federation mechanism for openstack—the classe view. *Concurrency and Computation: Practice and Experience*, 29(12):e4148, 2017. e4148 CPE-16-0379.

[29] Tejaswini Apte and Jatinderkumar R. Saini. A comprehensive and critical analysis of cross-domain federated identity management deployments. In Milan Tuba, Shyam Akashe, and Amit Joshi, editors, *ICT Systems and Sustainability*, pages 365–372, Singapore, 2021. Springer Singapore.

[30] Paul A. Grassi, Michael E. Garcia, and James L. Fenton. Nist: Digital identity guidelines. June 2017.

[31] Authlib - the ultimate python library in building oauth and openid connect servers. https://docs.authlib.org/en/latest/index.html. Accessed: 2021-10-15.

[32] Keycloack. https://www.keycloak.org/. Accessed: 2021-10-10.

[33] Keycloak - managing resource servers. https://www.keycloak.org/docs/latest/authorization_services/index.html#_resource_server_overview. Accessed: 2021-12-5.

[34] Sdn architecture. https://www.researchgate.net/figure/SDN-architecture-exclusive-the-management-plane_fig12_315382372. Accessed: 2021-09-23.

[35] Open network operating system. https://opennetworking.org/onos/. Accessed: 2021-09-30.

[36] Sandra Scott-Hayward. Trailing the snail: Sdn controller security evolution. 11 2017.

[37] Apache karaf. https://karaf.apache.org/. Accessed: 2021-10-3.

[38] Apache shiro. https://shiro.apache.org/. Accessed: 2021-10-3.

[39] Opendaylight authentication authorization & accounting. https://docs.opendaylight.org/projects/aaa/en/latest/dev-guide.html. Accessed: 2021-09-27.

[40] Ramachandra Kamath Arbettu, Rahamatullah Khondoker, Kpatcha Bayarou, and Frank Weber. Security analysis of opendaylight, onos, rosemary and ryu sdn controllers. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 37–44, 2016.

[41] Slf4j. https://www.slf4j.org/. Accessed: 2021-10-5.

[42] H2 database. https://www.h2database.com/html/main.html. Accessed: 2021-10-25.

[43] Opendaylight architecture. https://www.opendaylight.org/wp-content/uploads/sites/14/2019/03/OpenDaylight-Architecture-.png. Accessed: 2021-10-4.

[44] Lighty.io. https://lighty.io/. Accessed: 2021-09-29.

[45] Lighty.io authentication authorization & accounting. https://lighty.io/aaa-integration/. Accessed: 2021-09-29.

[46] Open network operating system oauth provider. https://wiki.onap.org/display/DW/OAuth+Provider+Implementation. Accessed: 2021-09-30.

[47] Open network operating system honolulu. https://wiki.onap.org/display/DW/SDN-R+Release+Honolulu. Accessed: 2021-09-30.

[48] Rest. https://restfulapi.net/. Accessed: 2021-10-15.

[49] Rest red hat. https://www.redhat.com/en/topics/api/what-is-a-rest-api. Accessed: 2021-10-15.

[50] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Péter Vörös and Attila Kiss. Security middleware programming using p4. In Theo Tryfonas, editor, *Human Aspects of Information Security, Privacy, and Trust*, pages 277–287, Cham, 2016. Springer International Publishing.

[52] James Yu and Imad Al Ajarmeh. An empirical study of the netconf protocol. In *2010 Sixth International Conference on Networking and Services*, pages 253–258, 2010.

[53] Ieee xplore. `https://ieeexplore.ieee.org/Xplore/home.jsp`. Accessed: 2021-9-27.

[54] Research gate. `https://www.researchgate.net/`. Accessed: 2021-9-27.

[55] Elsevier. `https://www.elsevier.com/`. Accessed: 2021-9-27.

[56] Academia. `https://www.academia.edu/`. Accessed: 2021-9-27.

[57] Google scholar. `https://scholar.google.com/`. Accessed: 2021-9-10.

[58] Keycloack server administration guide. `https://www.keycloak.org/docs/latest/server_admin/index.html#configuring-realms`. Accessed: 2021-10-1.

[59] Fernet. `https://cryptography.io/en/latest/fernet/`. Accessed: 2021-11-2.

[60] Docker. `https://www.docker.com/`. Accessed: 2021-10-26.

[61] Locust. `https://locust.io/`. Accessed: 2021-11-10.

[62] Elasticsearch. `https://www.elastic.co/pt/`. Accessed: 2021-11-10.

[63] Keycloak admin rest api. `https://www.keycloak.org/docs/latest/server_development/index.html#admin-rest-api`. Accessed: 2021-12-22.

[64] Sdn series part six. `https://thenewstack.io/sdn-series-part-vi-opendaylight/`. Accessed: 2022-03-29.

[65] Prometheus. `https://prometheus.io/`. Accessed: 2022-5-5.

[66] Nicolas Broders, Célia Martinie, Philippe Palanque, Marco Winckler, and Kimmo Halunen. A generic multimodels-based approach for the analysis of usability and security of authentication mechanisms. In *International Conference on Human-Centred Software Engineering*, pages 61–83. Springer, 2020.

Appendices

This page is intentionally left blank.

Appendix A

BIANFE: Object identification and authentication in federated scenarios

BIANFE: Object identification and authentication in federated scenarios

Carolina Gonçalves
University of Coimbra, CISUC, DEI
Coimbra, Portugal
mariapg@student.dei.uc.pt

Bruno Sousa
University of Coimbra, CISUC, DEI
Coimbra, Portugal
bmsousa@dei.uc.pt

Nuno Antunes
University of Coimbra, CISUC, DEI
Coimbra, Portugal
nmsa@dei.uc.pt

***Abstract*—Federated Identity Management enables convenient mechanisms to authenticate users and to authorize services, applications to specific users' resources. SAML and OpenID Connect that relies on OAuth 2.0 are commonly employed to enable Single-Sign-On features. Despite their wide usage in several domains (enterprise, web applications) they only aim to identify entities like persons and do not consider the different trust levels that a person can have with its devices, or even with the services provided by organisations participating or not in federated scenarios. BIANFE stands as a proposal for object identification and authentication in federated and non federated scenarios, considering the trust relations between end-users and the applications/services running in its devices. As work in progress, BIANFE tackles primarily the identification issue for objects, considering interoperability and privacy issues.**

***Index Terms*—Federated Identity Management, OpenID Connect, UUID, OAuth 2.0, Object**

I. Introduction

Federated Identity Management (FIM) has attracted the research community and enterprises to build several solutions, such as [1]: Kerberos, the Security Assertion Markup Language (SAML), Shibboleth, Open Authentication (OAuth 2.0), and OpenID Connect (OIDC.

Identity management solutions include mechanisms and architectures to exchange identity information between organisations that are federated for authentication purposes. Solutions such as SAML are mainly suited for enterprise contexts to enable Single-Sign-On (SSO), while OIDC is more generic being also suited for generic contexts [2]. OAuth 2.0 only supports authorisation of resources, while SAML and OIDC also support authentication of users.

The identity information in FIM solutions is commonly referred as claims or SAML assertions, and contains user unique identifiers, Private Identifiable Information (PII) items, such as email address, phone number. The major issue with existent FIM solutions is that they aim to identify entities like persons and do not consider the different trust levels that a person can have with its devices, or even with the services provided by organisations participating or not in federated environments.

The Object Identifier - OID is commonly employed for naming nodes in the Internet, to name objects in X.509 certificates, as identifiers in databases, in management protocols like SNMP [3]. The Universal Unique Identifier - UUID is employed to identify uniquely resources, for instance to provide unique values for primary keys in tables, or to identify objects in operating systems.

The Uniform Resource Name (URN) can be combined with other mechanisms, like the OID [4], and UUID [5]) to convey identifiers in a standard fashion and to provide interoperability between heterogeneous systems.

BIANFE is as a proposal for object identification and authentication in federated scenarios, considering the trust relations between end users and the applications running in its devices. BIANFE tackles primarily the identification issue for objects, considering interoperability issues. This paper presents the preliminary approaches being considered in BIANFE. The contributions of BIANFE are: 1- A proposal to identify objects in environments, where multiple organizations are set up in a federated mode (i.e. relying on a common identity provider); 2- A proposal to authenticate, authorise resources considering the trust relations between persons and objects/services; 3- An approach compatible with OIDC to enable the identification of distinct entities.

II. BIANFE: Objects identification and authentication

BIANFE considers *'objects'* as an entity that can be physical (e.g., device) or virtual (e.g., application), with associated environments (i.e., mapping to a specific operating system or virtualization platform), on which user agents are executed. User agents, or web browsers, make the interface with the end-user, to gather its consent on authorizing the access from a service to a specific resource (e.g., email address).

A. Problem statement & Use case

The use case described in Fig. 1 illustrates an user with multiple devices on which user agents (UAs) can connect to diverse services. UAs can run on distinct environments of a device, which can represent the operating systems (e.g., Windows, Linux), where a browser (e.g. Firefox) runs. The environment can also be associated with virtualised contexts. For instance, a user agent based on the `curl` tool can run inside a docker container, or even run as a microservice in kubernetes (K8s).

The user has different trust levels regarding its devices, and with the respective user agents. A simple example can consider

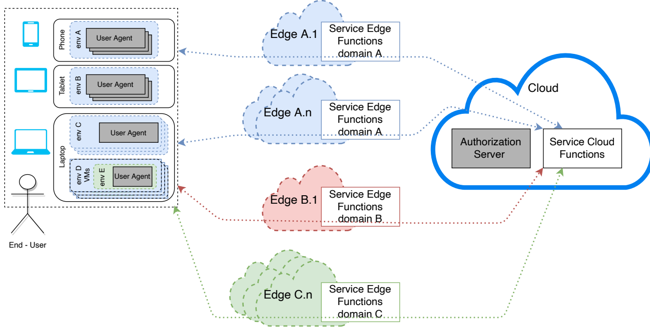


Fig. 1. Use case and associated problem statement

the Eve user *person:Eve* that has an Android mobile phone - *device:android* and an iPhone device - *device:iphone*. Eve may tend to trust more on the iPhone device given the full control of Apple in terms of hardware and software.

B. Identification of objects with URN and UUIC

The URN, as per RFC 4122 [5] is a feasible solution to identify objects, persons and the respective trust relations. In particular, by allowing an user to specify generic trust relations (i.e trusting in a device), or providing detailed information regarding the environment on which the user agent runs (i.e., trusting in a specific device and the windows operating system).

The advantage of the URN is the human interoperability. For instance, the person Eve to identify the trust of the Google Chrome in the android device could use the URN in the listing 1 at line 1, while the URN in line 3 identifies the Google Chrome user agent running in the iPhone device.

Listing 1. URNs for Google Chrome in android and iPhone devices

```
urn:person:Eve:device:android:userAgent:googleChrome

urn:person:Eve:device:iphone:userAgent:googleChrome
```

A more granular identification can be formulated by including the detailed information of the environment, for instance, for the the cloud device B, as illustrated in listing 2. The details in this example include the identification of the K8s cluster and POD (group of containers).

Listing 2. URN with information of the environment

```
urn:person:Eve:device:cloudDevB:environment:k8sProd:
    subenvironment:PODA:subenvironment:dockerA:userAgent
    :PythonRequests
```

UUID supports multiple options to generate unique identifiers [5]: UUID v1 generates identifiers based on the Medium Access Control - MAC address of an host, the date and time values and a random value; UUID v3 and v5 provide the means to generate unique identifiers relying on a namespace identifier (e.g., domain name) and on generic names assigned by the user; UUID v4 provides unique identifiers relying solely on random numbers.

Both UUID v1 and v4 are not suitable for the purpose of BIANFE due to the randomness effect. In addition, the MAC address of a device is not considered a reliable identifier [3]. The main difference between UUID v3 and v5 relies on the hashing algorithm, where v5 stands as the most secure option by employing the SHA-1 algorithm.

UUID v5 requires two parameters to generate a unique identifier, the namespace and a string with the name assigned by the end user. The namespace is commonly represented in the form a UUID (e.g., `306bb302-bc09-438a-9a49-21f7b02f3060`), or can rely on pre-defined identifiers for fully qualified domain names (DNS), URLs, OID or X.500 distinguished names. The listing 3, illustrates how the UUID can be generated, via the `genBianfeID()` function, for the user agent running in a environment and on a specific device.

Listing 3. Example of UUID for the Google Chrome

```
# See https://docs.python.org/3/library/uuid.html
import uuid
nameUA='urn:person:Eve:device:android:userAgent:
    googleChrome'
nameSpace= uuid.NAMESPACE_DNS
def genBianfeID(nameSpace, nameUA):
    bianfeID=uuid.uuid5(uuid.NAMESPACE_DNS, urnUA)
    return bianfeID
```

III. Conclusions & Next Steps

BIANFE stands as a proposal for object identification and authentication in federated scenarios, considering the trust relations between end users and applications/services running in the devices. BIANFE tackles primarily the identification issue for objects and persons by proposing an approach that relies on URN and UUID standards. The next steps for BIANFE include its integration with the OIDC standardised flows (e.g. Authentication Code flow), to include the BIANFE identifiers in scenarios with services provided in federated environments.

Acknowledgment

This work is funded by project AIDA (POCI-01-0247-FEDER045907), co-financed by the European Regional Development Fund (ERDF) through the Operacional Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal, and by National Funds through the Portuguese funding agency FCT - Fundação para a Ciência e a Tecnologia with grant SFRH/BD/129771/2017 and within project UIDB/50014/2020.

References

[1] A. Rasiwasia, "A Framework To Implement OpenID Connect Protocol For Federated Identity Management In Enterprises," Ph.D. dissertation, 2017.
[2] N. Naik and P. Jenkins, "Securing digital identities in the cloud by selecting an apposite Federated Identity Management from SAML, OAuth and OpenID Connect," in *International Conference on Research Challenges in Information Science (RCIS)*. IEEE, may 2017, pp. 163–174.
[3] D. R. E. Downs and et al., "On the utility of identification schemes for digital earth science data: An assessment and recommendations," *Earth Science Informatics*, vol. 4, no. 3, pp. 139–160, 2011.
[4] M. H. Mealling, "A URN Namespace of Object Identifiers," RFC 3061, Feb. 2001.
[5] P. J. Leach, R. Salz, and M. H. Mealling, "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122, Jul. 2005.