MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Adaptive Authentication in Keycloak

Master's Thesis

BC. MARTIN BARTOŠ

Brno, Spring 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Adaptive Authentication in Keycloak

Master's Thesis

BC. MARTIN BARTOŠ

Advisor: prof. RNDr. Václav Matyáš, M.Sc., Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2024

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, AI-driven tools were employed, including *Grammarly* for grammatical analysis and *ChatGPT* for occasional textual expression enhancements. I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Bc. Martin Bartoš

Advisor: prof. RNDr. Václav Matyáš, M.Sc., Ph.D.

Acknowledgements

I am deeply grateful to my supervisor, Václav Matyáš, for his guidance and support throughout this thesis. I would also like to thank Marek Posolda for his valuable insights, which greatly enriched this work. Their contributions have been instrumental in its completion.

Abstract

Keycloak is an open-source identity and access management system that provides robust security solutions for applications and services. It acts as a centralized authentication and authorization platform, enabling organizations to secure their resources while offering Single Sign-On capabilities. Keycloak is widely used in various industries, including web applications, microservices, and APIs, to protect sensitive data and streamline user access control.

The thesis aims to enhance the security and user experience of authentication processes in Keycloak. Traditional static authentication mechanisms, such as username and password, are vulnerable to various security threats, including password breaches, phishing attacks, and unauthorized access. Adaptive authentication might help mitigate these issues.

Adaptive authentication aims to improve the security posture by dynamically adjusting the authentication requirements based on contextual factors. That means the system can adapt its authentication methods and policies in response to changing circumstances.

Keywords

adaptive, authentication, Keycloak, open-source, CNCF, risk-based, security

Contents

1 Introduction **1**

2 Adaptive Authentication **3**
2.1 Concept 3
2.2 Benefits 3
2.3 Challenges 4
2.4 Authentication Factors 5
2.4.1 Knowledge Factor (Something you know) . . . 6
2.4.2 Possession Factor (Something you have) 6
2.4.3 Inherence Factor (Something you are) 7
2.4.4 Location Factor (Somewhere you are) 7
2.4.5 Behavior Factor (Something you do) 8
2.5 Risk Levels 8

3 Existing Solutions **9**
3.1 Okta 9
3.2 Citrix 13
3.3 OneLogin 14
3.4 Silverfort 15
3.5 Beyond Identity 17

4 Keycloak **20**
4.1 Introduction 20
4.2 Service Provider Interface 20
4.2.1 Necessary Parts 21
4.2.2 Configuration 22
4.3 Authentication Flow 22
4.4 Authentication Execution 23
4.5 Basic Authentication Flow 24

5 Specification **26**
5.1 Extendability 27
5.2 Basic Implementation 27
5.3 Policy Engine 28
5.4 Artificial Intelligence 28
5.5 External Integration 29

6 Design **30**
6.1 User Context 30
6.1.1 Specific User Context 31
6.1.2 User Context Condition 33
6.2 Authentication Policies 36
6.2.1 Authentication Flow 37
6.2.2 Authentication Policy Authenticator 37
6.2.3 Policy Evaluation Example 38
6.3 Risk-based Authentication 41
6.3.1 Risk Evaluators 42
6.3.2 Risk Engine 45
6.3.3 Risk Levels 50
6.4 Artificial Intelligence 54
6.5 External Integration 56

7 Implementation **57**
7.1 Authentication Policy 57
7.1.1 Data Access Object 58
7.1.2 REST API 62
7.2 User Context 64
7.3 Risk Evaluator 66
7.3.1 Configuration 67
7.4 Risk Engine 68
7.4.1 Asynchronous Processing 69
7.4.2 Retry Evaluations 71
7.4.3 Evaluations Timeout 72
7.4.4 Evaluation 72
7.5 Artificial Intelligence 73
7.6 External Integration 76

8 Conclusion **78**

Bibliography **80**

List of Figures

3.1 Okta Authentication Policy Rule Conditions. 12
3.2 Silverfort Adaptive Policy 17
3.3 Beyond Identity Authentication Policies 19

4.1 WebAuthn Flow. 25

6.1 User Context Diagram. 31
6.2 IP Address User Context Diagram. 32
6.3 User Context Condition Diagram. 33
6.4 Operation Diagram. 34
6.5 IP Address Context Condition. 35
6.6 Authentication Policy Browser Restriction. 39
6.7 Browser Restriction Condition. 40
6.8 Browser Restriction Deny Access. 40
6.9 Risk Evaluator Diagram. 42
6.10 Risk Evaluators Configuration. 44
6.11 Risk Engine Diagram. 47
6.12 Risk Engine Configuration. 48
6.13 Stored Risk Provider Diagram. 48
6.14 Risk Engine Authenticator Configuration. 49
6.15 Risk Level Diagram. 50
6.16 Default Risk Levels. 51
6.17 Simple Risk Level Configuration. 52
6.18 Simple Risk Level Flow. 53
6.19 AI NLP Engine Diagram. 54
6.20 Risk Evaluator and AI NLP Engine Diagram. 55
6.21 Remote Location Context. 56

7.1 Authentication Policies Parent. 57
7.2 Authentication Policy Provider. 58
7.3 Add Policy Method. 60
7.4 Get All Authentication Policies. 61
7.5 Get All Authentication Executions. 61
7.6 Authentication Policies REST API. 62
7.7 Authentication Policies REST API Implementation. 64
7.8 Extend *UserContext* Interface. 64
7.9 Implementation of the *IpProxyContext*. 65

7.10 Login Failures Risk Evaluator. 66
7.11 Login Failures Risk Evaluator Evaluations. 67
7.12 Login Failures Risk Evaluator Configuration. 68
7.13 I/O Threads. 70
7.14 Asynchronous Processing Flow. 71
7.15 Risk Evaluators Retry Mechanism. 71
7.16 Risk Evalutors Timeout. 72
7.17 Risk Engine Evaluation. 73
7.18 OpenAI Integration. 74
7.19 AI NLP Context Message. 75
7.20 AI NLP Device Query. 75
7.21 ChatGPT Device Response. 76
7.22 AI NLP Login Failures Query. 76
7.23 External Service *ipapi*. 77

1 Introduction

In today's digital landscape, where cybersecurity threats continue to evolve in sophistication and frequency, the need for robust authentication mechanisms to safeguard sensitive data and systems has become critical. Authentication incorporates two primary domains: user authentication and data authentication. User authentication assures the identity verification of an individual to grant access to a system or service. It ensures that individuals are who they claim to be before allowing them access to resources. Data authentication, on the other hand, assures data integrity and origin, ensuring it remains unaltered during transmission or storage, focusing solely on data content.

Adaptive authentication extends user authentication, as it is an advanced security mechanism designed to dynamically adjust the level of authentication based on various factors such as user behavior, context, and risk levels. Unlike traditional authentication methods that rely on static credentials or fixed Multi-Factor Authentication (MFA) processes, adaptive authentication continuously evaluates the risk associated with each authentication attempt and adapts the authentication requirements accordingly.

Adaptive authentication offers significant advantages by evaluating various factors such as user behavior, device characteristics, location, and risk attributes to provide a more secure and seamless user experience. Adaptive authentication enhances security without compromising convenience by dynamically adjusting authentication requirements based on real-time assessments.

This thesis is aimed at the principles and implementation of adaptive authentication within Keycloak[1], an open-source Identity and Access Management (IAM) system. Keycloak is widely used to secure web applications, microservices, and APIs by providing centralized authentication and authorization services with a broad user community.[1] Keycloak is one of the leading open-source IAM solutions on the market and is getting increasingly popular due to its capabilities and rapid development. Extending its capabilities further by integrating adaptive authentication will have a magnificent impact

1. https://www.keycloak.org/

on enhancing security, improving user experience, and meeting the dynamic needs of modern enterprises.

The roadmap of this thesis is as follows – Chapter 2 introduces adaptive authentication, outlining its benefits and challenges. It explains how adaptive authentication dynamically adjusts based on various factors. Chapter 3 reviews and compares existing market solutions, providing a comprehensive understanding of different implementations and their pros and cons. Chapter 4 focuses on Keycloak, detailing its architecture, service provider interface, and authentication processes. Chapter 5 specifies the requirements and goals for this topic. Chapter 6 covers design aspects such as user context, authentication policies, risk-based authentication, and AI integration, offering a development blueprint with diagrams and examples. Chapter 7 presents the actual implementation, describing the development itself with the necessary coding and configuration for adaptive authentication in Keycloak.

2 Adaptive Authentication

2.1 Concept

At its core, adaptive authentication usually leverages real-time data analysis and contextual insights to make informed decisions about the appropriate level of authentication needed to verify a user's identity. This dynamic approach allows organizations to strengthen security measures while minimizing disruption to user workflows. Unlike static authentication methods, which apply the exact authentication requirements to all users and sessions, adaptive authentication systems continuously evaluate risk factors and adjust authentication measures accordingly.

By leveraging real-time data and analytical capabilities, adaptive authentication enhances security while also optimizing the user experience.[2] Adaptive authentication is closely tied to the Identity Thread Detection and Response (ITDR) security discipline. Certain aspects of adaptive authentication are fundamental concepts of the ITDR strategies aimed at identifying and responding to identity threats.

2.2 Benefits

As was stated in the introduction of adaptive authentication, the main benefits behind its usability are mainly the dynamics. As in the traditional approaches, the static MFA is very often used when assessing authentication policies with predefined exact rules that the user might comply with or the system state is in.

Adaptive authentication may provide the ability to have a more frictionless approach when a user is trying to prove their identity. [3, 4] The summarization of the more exciting benefits might be as follows:

- **Strengthen security** – when a user is trying to access sensitive or important resources, additional factors might be required.
- **Context–aware protection** – takes into account contextual information, such as device information, location, IP address, and behavioral patterns.

- **Flexibility and extendability** – by leveraging more contextual data, the policies and rules might better comply with customers' needs.
- **Better user experience** – may provide more frictionless authentication as a user is not required to provide too many aspects of their identity.
- **Integration with AI/ML** – artificial intelligence, or machine learning, might be used for processing contextual data, assessing risk levels, and making more knowledgeable decisions.
- **Alignment with zero trust model** – Adaptive authentication, in most cases, aligns well with the zero trust security model due to its main principle to 'never trust, always verify'. [5]

2.3 Challenges

While the benefits of adaptive authentication are clear and compelling, it is essential to acknowledge that this approach might face several challenges. Despite its dynamic nature and improved user experience, adaptive authentication presents several challenges that organizations must address to realize its full potential.

The summarization of the biggest challenges might be as follows:

- **Complexity** – the implementation of the processing user behavior and evaluation of contextual data might be very complex and time-consuming.
- **Accuracy of risk assessment** – as the contextual data might not be reliable and correct in a particular context, the risk evaluation needs to be properly executed – we need to prevent the count of false positives and false negatives.
- **Privacy and data protection** – as the contextual data might help to strengthen security, these data need to be collected and processed – compliance with data protection regulations and users' preferences need to be assessed.

- **Potentially non-deterministic** – the benefit of integration of AI/ML might also bring some challenges as we can achieve non-deterministic behavior in processing contextual data and assessing risk levels – it might be challenging to predict possible outcomes, which also makes testing harder.[2, 3, 4]

2.4 Authentication Factors

Authentication factors play a crucial role in verifying the identity of users and granting them access to digital systems, applications, and data. In the realm of cybersecurity, authentication factors serve as the building blocks of authentication mechanisms, providing layers of security to protect against unauthorized access and mitigate the risk of security breaches.

Authentication factors exist to address the inherent limitations and vulnerabilities of traditional password-based authentication methods. While passwords have long been the primary means of authenticating users, they are susceptible to various security threats, such as brute-force attacks, phishing, and password reuse. Authentication factors enhance security by introducing additional layers of verification beyond mere knowledge of a password.[6]

Adaptive authentication uses authentication factors to evaluate the potential risk that a user accessing a particular resource is fraudulent.

These factors use contextual data obtained by arbitrary approaches and can be divided into a few categories:

1. **Knowledge factor** – Something you know.
2. **Possession factor** – Something you have.
3. **Inherence factor** – Something you are.
4. **Location factor** – Somewhere you are.
5. **Behavior factor** – Something you do.

The first three authentication factors *Knowledge factor*, *Possession factor*, and *Inherence factor* are the well-known traditional factors used

in many applications managing identity access. The last two authentication factors *Location factor*, and *Behavior factor*, are extensions of traditional approaches and are extensively used for adaptive authentication.[7]

2.4.1 Knowledge Factor (Something you know)

Knowledge factors involve users providing specific information or data to access a secured system. The most common type of knowledge-based authentication is through passwords or personal identification numbers (PINs), which are used to limit system access.

Typically, in generic applications or network logins, users need to provide both a username/email and its associated password or PIN to gain entry. It is important to note that solely providing a username or email does not serve as an authentication factor. Instead, it functions as the user identification within the system.

Authentication occurs when the provided password or PIN verifies the associated username or email, confirming the identity of the user.

2.4.2 Possession Factor (Something you have)

Possession factors require users to possess a specific physical object before gaining access to the system. Typically, possession factors are managed through a device known to belong to the correct user. A typical process flow for possession-based authentication might look like the one described in the following lines.

The user registers an account with a password, and their phone number is recorded at the time of registration. The user logs in to their account with the username and password. When the user requests access to the system, a one-time password is generated and sent to the mobile phone number of the user. The user enters the newly generated one-time password and gains access to the system.

One-time passwords can be generated by a device like the RSA SecurID, or they may be automatically generated and sent to the user's cellular device via SMS. In either case, the correct user must possess the device that receives or generates the one-time password to access the system.

2.4.3 Inherence Factor (Something you are)

Inherence factors authenticate access credentials based on unique user characteristics. These include fingerprints, thumbprints, palm or handprints, as well as voice and facial recognition and retina or iris scans.

When systems proficiently identify users via their biometric data, inherence stands as one of the most secure authentication methods. However, a downside is the potential loss of flexibility in account access.

For instance, a system requiring a fingerprint scan can only be accessed on devices equipped with the necessary hardware. While this restriction enhances security, it may diminish user convenience. [8]

2.4.4 Location Factor (Somewhere you are)

Location factors allow administrators to implement geolocation security checks to verify user location before granting access to an application, network, or system.

Consider a multinational corporation with offices in various cities worldwide. In this scenario, a security analyst might identify a user attempting to access the network from an IP address originating from a country different from their assigned office.

Geolocation security ensures access is restricted to users within a specified geographic area. While IP addresses offer insight into network traffic origin, hackers can avoid detection by using Virtual Private Networks (VPN) to mask their location.

Alternatively, Medium Access Control (MAC) addresses, unique to each computing device, can serve as a location-based authentication factor, allowing system access only from authorized devices within a defined range.

2.4.5 Behavior Factor (Something you do)

Behavior factors depend on user actions to access the system. In systems accommodating behavior-based authentication factors, users may configure a password by executing specific actions within a defined interface, which they can subsequently replicate to verify their identity.

Consider keystroke dynamics and mouse movement patterns as examples of behavior-based authentication factors. Keystroke dynamics can study the unique way a user types on a keyboard. Mouse movement patterns can analyze the way a user moves the mouse while interacting with a system. [6, 9, 10]

2.5 Risk Levels

Risk levels in adaptive authentication typically refer to the likelihood that a particular access attempt is fraudulent or unauthorized. These risk levels are often categorized into low, medium, and high, though the specific criteria for each level can vary depending on the implementation.

Low-risk access attempts might involve familiar devices, typical login times, and consistent behavior patterns. In these cases, the authentication process may be streamlined with minimal additional verification steps.

Medium-risk access attempts might involve slightly unusual behavior, such as logging in from a new location or using a different device. In these cases, the system may prompt for additional verification, such as a one-time password sent via email or SMS.

High-risk access attempts might involve highly suspicious behavior, such as multiple failed login attempts, login from a known compromised device, or access from a location far from the user's usual ones. In these cases, the system may require stricter authentication measures.

3 Existing Solutions

This specific part of the thesis is focused on the analysis of existing solutions provided by particular vendors that are present on the market. It is very beneficial to be aware of solutions for adaptive authentication as there are already some companies and institutions that are trying to solve the same problem as is touched on in this thesis.

The intention behind analyzing existing solutions is to get information from the other vendors in order not to reinvent the wheel. In order to create an optimal solution for adaptive authentication, it is advantageous to recognize common patterns in these solutions, aggregate the best parts, enhance them, and tailor them to the needs of Keycloak.

In the following sections, there are some brief descriptions of products provided by the mentioned vendors. The list of existing solutions and their ordering is not sorted by any particular keys but rather gathered as the most promising and exciting solutions.

3.1 Okta

Okta is an identity management company that provides cloud-based solutions for businesses to connect people and technology securely. Okta offers a platform that enables organizations to manage and secure user authentication, access control, and identity governance across various applications and devices. The company's services include single sign-on, multi-factor authentication, lifecycle management, and Application Programming Interface (API) access management. Okta describes its adaptive and risk-based capabilities as follows:

"User and risk levels are always changing; your security should be able to keep up. Okta Adaptive MFA allows for dynamic policy changes and step-up authentication in response to changes in user and device behavior, location, or other contexts."[11]

Adaptive MFA supports detection and authentication challenges for riskier situations like:

- Use of weak/breached passwords.
- Proxy use.
- Geographic location and zone changes.
- Brute force and denial-of-service attacks.
- Use of new/untrusted devices.
- Indications of anomalous behavior.

ThreatInsight Service

Okta approach introduces the consideration of external risk contexts, enhancing organizations' understanding of authentication processes by utilizing the ThreatInsight service. This service collects information about the origin of sign-in activity directed at Okta endpoints.

ThreatInsight evaluates authentication requests by analyzing data to identify potentially suspicious activity by using machine learning and provides reasonable threat detection.

This service furnishes critical details encompassing users, devices, locations, and networks, thereby facilitating more informed decision-making regarding access permissions.

ThreatInsight provides a way to add risk factors from external sources like third-party services. However, relying on remote services may introduce challenges, such as potential latency issues, which could impede the authentication process and lead to user frustration and operational inefficiencies.[12, 13]

Ensuring the safety and trustworthiness of these outside sources is very important. I think leveraging the risk factors from external sources must be carefully considered in order to avoid the risk of data leaks or mistakes that could harm the authentication system.

System Log Events

As the Okta ThreatInsight service is able to collect information about the origin of the sign-in activity, it can identify the threat and follow up with proper actions. One of these actions is recording the potentially malicious requests and their origins into a system log.

It can record a variety of events, such as sign-in attempts from suspicious IP addresses or detection of security threats. Moreover, it can record information about the continuity of potential cyber-attacks involving the attack duration and aggregation of relevant events.

With the gathered data, an administrator can easily create a blocklist of IP addresses with denied application access. The system log events contain information about how the risk level was determined for each authentication attempt. It can take into consideration factors like anomalous location, anomalous device, or suspected threat based on Okta ThreatInsight detection.[14, 15]

In all cases, such comprehensive information about login attempts and their metadata is necessary for followed risk factor evaluation and success of adaptive authentication.

Authentication Policies

Each application specified in Okta comes with its own authentication policy. This policy ensures that people trying to log in to the application meet certain conditions and follow specific rules for verification.

While some conditions may be similar between authentication policies and global session policies, they have different jobs. If someone gains access to Okta through the global session policy, it does not mean they automatically have access to all their apps. You can make a unique policy for each application or use a few policies for multiple apps. There are also preset policies in Okta for apps with standard sign-on requirements.

Every authentication policy has a set of rules specifying the authentication flow behavior during an authentication attempt. In every rule, you can define various conditions, and if all are evaluated as true, the authentication settings with proper actions are applied.

The conditions might consist of the setting of user role, user group, device state, device platform, and much more. When these conditions are met, actions like explicit access denial or step-up authentication are done. For the step-up authentication, you can specify what authentication mechanism should be used, such as enforcing MFA with specific attributes of the possession factor.

The specific rule conditions might look like in Figure 3.1.

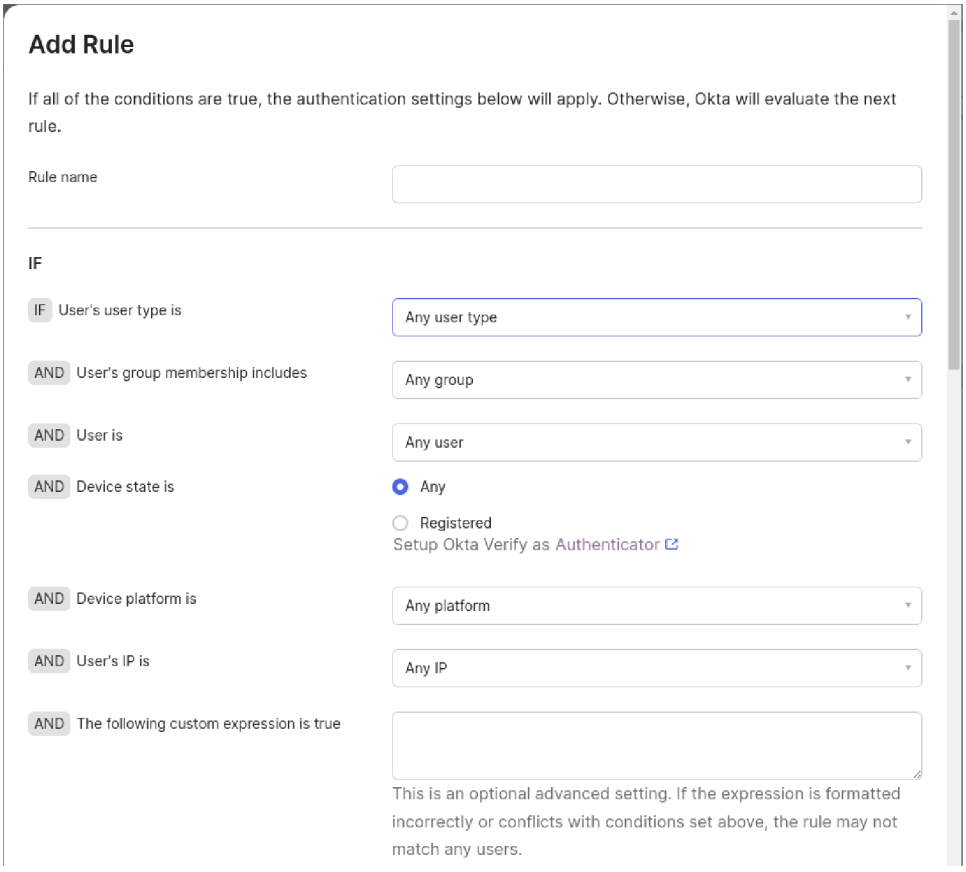

Figure 3.1: Okta Authentication Policy Rule Conditions.

3.2 Citrix

Citrix Systems, Inc. is an American multinational cloud computing and virtualization technology company that provides server, application, and desktop virtualization, networking, software as a service (SaaS), and cloud computing technologies. Citrix products were claimed to be used by over 400,000 clients worldwide, including 99% of the Fortune 100 and 98% of the Fortune 500.

Takeaways

I appreciate how the Citrix adaptive authentication solution handles various details of user authentication. It carefully considers factors such as user roles, organizational affiliations, and unique user attributes. These factors are crucial in assessing the level of risk in each authentication attempt. The solution does not treat each factor in isolation.[16, 17]

Instead, it integrates all these details to determine the overall security of an authentication attempt. This enhances overall security and improves the user experience by making the authentication process smoother by considering the whole context of risk factors.

The other valuable feature that Citrix provides is called Device Posture, as it allows to specify requirements for devices that are accessing the application. It enhances authentication policies in such a manner that authentication rules can be comprehended about the device attributes.

Authentication policies are necessary parts of the adaptive authentication initiative as they provide the capability to adapt to the surrounding environment. You can specify the version of the operating system, geolocation, network location, browser, or whether or not the antivirus is enabled on the device. It is also possible to integrate the device posture with third-party services as the evaluation can be shifted there.

The device posture can be used for the individual authentication policies, but even also for the risk factors evaluation. We can assess the risk evaluation mechanism to properly reflect particular cases of device posture and contribute to the overall risk evaluation of the adaptive authentication.[18]

3.3 OneLogin

OneLogin, Inc. is a cloud-based identity and access management (IAM) provider that develops a unified access management (UAM) platform for enterprise-level businesses and organizations. [19]

Takeaways

I like how OneLogin describes risk-based authentication with the leverage of machine learning. Such solutions learn from user behavior over time to create a detailed profile of each login habit of the user. That includes tracking devices, typical login times, and usual work locations. Additional analysis of IP addresses, network reputations, and threat data is used to enhance security further.

Solution SmartFactor, provided by OneLogin, gathers risk factor values described above, which are used as inputs for their Vigilance AI solution. After analyzing these factors via the artificial intelligence engine, the contextual risk score is evaluated. The risk score is in the range 0-100. Based on the evaluated score, appropriate actions are executed, such as requiring additional authentication steps or denying access.[19]

Vigilance AI

Vigilance AI also provides training API, where the AI models are trained, and the contextual risk score evaluation can be continuously improved. As the complex contextual data is sent to the Vigilance AI service, the response contains additional information about triggers, which are indicators of some of the key items that influenced the risk score. It brings a better overview and addressability of the risk score evaluation in a human-readable format.

Vigilance AI supplies an API for the creation of a custom rule to gain more control over the risk scoring of events. Administrators are able to create rules based on the allow/deny list of IP addresses or current country locations. [20]

Statistics

An exciting thing is furthermore the gathering of risk score evaluation statistics, as administrators can obtain the count of sign-in events grouped by their risk score bucket. Vigilance AI divides the risk score bucket, previously called risk levels, into five categories: *minimal*, *low*, *medium*, *high*, and *very high*.

To be more specific, the statistics contain the number of sign-in events for each particular risk level, which might be beneficial for following data analysis. [21, 22]

3.4 Silverfort

Silverfort, Inc. provides several security-related products, such as the Advanced MFA, Identity Threat Detection and Response solution, Risk-based authentication, and others. Silverfort was founded in 2016 by Hed Kovetz, Yaron Kassner, and Matan Fattal. It is headquartered in Tel Aviv and has offices in the US and Singapore, as well as local Sales teams in over 18 countries across EMEA, NA, and APAC.[23]

Takeaways

Silverfort provides risk-based authentication with the possibility of specifying adaptive policies. I like the concept of adaptive policies, as it is possible to create custom authentication policies to meet unique requirements for the authentication process.

These policies can leverage risk scoring provided by the Silverfort solution and trigger proactive security controls, such as requiring additional authentication steps or denying access. This concept is very flexible and provides the possibility to explicitly set particular risk indicators in order to react to specific attacks that your environment is likely to experience.

You can see an example of configuring the described adaptive policy in Figure 3.2 below. This adaptive policy is basically a rule for the authentication process, and when conditions of the adaptive policy are met, a particular action is executed.[23]

Risk Factors Evaluation

The main takeaway from the solution is the understanding of the factors necessary for risk evaluations and their contextual tightness. Silverfort divides the risk factors into three main categories: access source, user, and authentication mechanism.

In assessing the access source, the security posture of the device is crucial. This includes evaluating factors such as the operating system version, the application of security patches, and the presence of antivirus software. The managed status of the device by an organization indicates a higher level of control and security measures.

Furthermore, detecting any malware or suspicious software on the device is essential to mitigate potential risks. Similarly, assessing the reputation of the network address or IP from which the authentication attempt originates, along with geolocation checks, helps identify any anomalies or potential risks.

User analysis focuses on past authentication attempts to establish a baseline of normal behavior. Deviations from this established authentication trail, such as sudden changes in behavior, unusual access patterns, or logins from unfamiliar locations, may indicate potential risks and warrant further investigation. Additionally, examining actions like interactive logins with service accounts or admin logins from unfamiliar devices helps identify suspicious behavior that could compromise security.

Lastly, the authentication mechanism involves analyzing abnormalities or vulnerabilities in the underlying authentication process. This includes identifying potential threats like pass-the-hash attacks, which could exploit weaknesses in authentication protocols, thereby enhancing overall security measures. [24, 25]

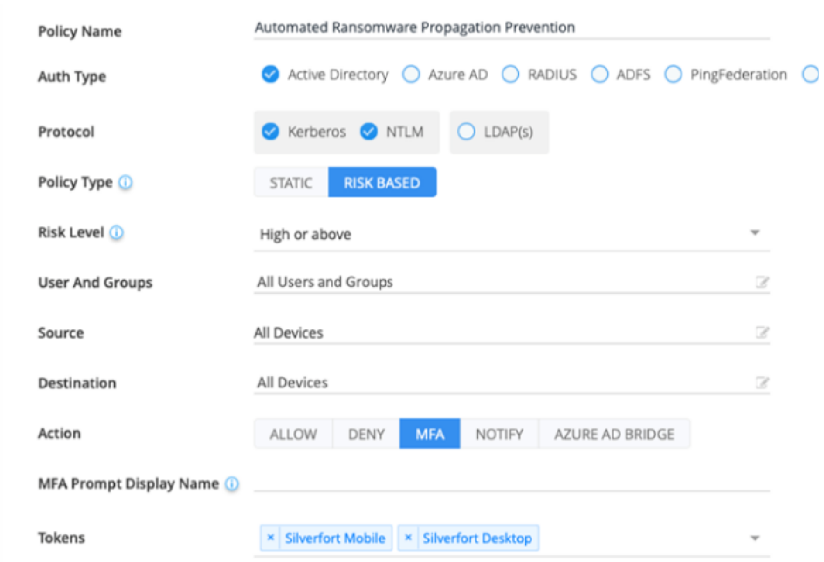

Figure 3.2: Silverfort Adaptive Policy. Source: [23].

3.5 Beyond Identity

Beyond Identity, Inc. is a company that specializes in passwordless identity management and authentication solutions. They offer a platform that aims to eliminate the need for passwords and provide more secure and user-friendly authentication methods.

Takeaways

I like the concept called the triad of risk that Beyond Identity provides in relation to adaptive (risk-based) authentication. The triad of risk is a methodology for conceptualizing risk factors when permitting or denying access to applications. The triad consists of three factors: *Device risk*, *Application risk*, and *Contextual risk*.

Device Risk involves assessing the security of the user hardware, considering factors like passcodes, authentication methods, operating system updates, the presence of firewalls, etc. This assessment helps determine the likelihood of the device being compromised.

Application Risk focuses on the potential impact an application access could have on operations if exploited by a malicious actor rather than the inherent riskiness of the application itself. It considers the sensitivity of the data or functions accessible through the application.

Contextual Risk takes into account the user behavior and environment, especially in the context of remote work. Factors such as geographic location, typical access patterns, and time of access are considered to identify abnormal behavior that may indicate fraud or malicious intent.

Adaptive authentication uses contextual factors to evaluate access attempts, triggering alarms for deviations from established patterns. Certain anomalies, like accessing an application from an unusual location, are given higher risk weighting.

Authentication and Device Policies

Beyond Identity also provides the feature of authentication and device policies. As was stated previously, authentication policies may provide the possibility to create custom authentication policies to meet unique requirements for the authentication process. Other than that, additional device policies provide an additional security layer, as devices that are trying to access the application need to meet the requirements specified by these policies.

You can notice that the user experience around configuring these policies varies as the user interface might have different representations, as shown in Figure 3.3. [26]

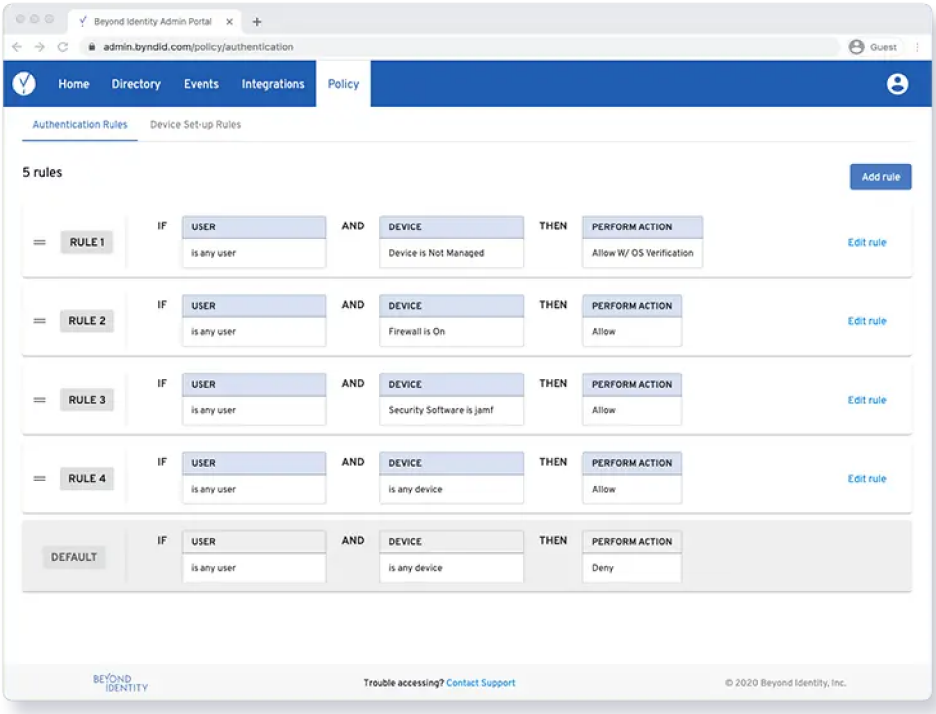

Figure 3.3: Beyond Identity Authentication Policies. Source: [26].

4 Keycloak

4.1 Introduction

Keycloak is a very popular open-source single sign-on and identity and access management solution for modern applications and services. The goal of Keycloak is to simplify security so that it is easy for application developers to secure the apps and services they have deployed in their organization. Security features that developers normally have to write for themselves are provided out of the box and are easily tailorable to the individual requirements of customer organizations.

Keycloak provides customizable user interfaces for login, registration, administration, and account management. Keycloak can be also used as an integration platform to hook it into existing Lightweight Directory Access Protocol (LDAP) and Active Directory servers. Moreover, Keycloak provides the delegation of authentication to third-party identity providers like Facebook and Google.

Since April 2023, Keycloak has been under the stewardship of the Cloud Native Computing Foundation (CNCF) as an incubating project.[1]

4.2 Service Provider Interface

Service Provider Interface (SPI) is an API intended to be implemented or extended by a third party. It can be used to enable framework extension and replaceable components.[27]

SPI is an essential tool for developers to customize different aspects of Keycloak as per their specific requirements. Using SPI, developers can create their own Keycloak extensions and modify functionalities such as the login screen appearance, adding new authentication methods, or integrating with different credential stores.

SPI allows developers to adapt Keycloak to diverse environments and business needs. This includes enhancing security measures, integrating with different databases, or improving the overall user experience. Essentially, SPI empowers developers to extend Keycloak beyond its original functionalities, making it adaptable and versatile across a range of scenarios.

4.2.1 Necessary Parts

In order to properly implement custom SPI for Keycloak, it is necessary to comply with the approach Keycloak suggests. The essential parts of the custom SPI are new classes that implement these interfaces:

- *Spi.*
- *Provider.*
- *ProviderFactory.*

A specific implementation of the *Spi* interface provides some sort of wrapper for the whole custom SPI representation with the required specification of provider class and provider factory class. Besides that, certain other attributes are present, such as name and flag, used to determine whether the SPI is internal or not.

A specific implementation of the *ProviderFactory* interface provides the functionality to create instances of the *Provider* classes. The *ProviderFactory* follows the Factory design pattern, and only one factory exists for the server. It supplies a shared configuration used for the creation of new providers, as it does not have to be evaluated or obtained for every single creation of the particular providers.

Additionally, users are able to change only the behavior for the creation of providers by implementing different provider factories, which brings more freedom to the development.

A specific implementation of the *Provider* interface provides an actual object with the capability of bringing a particular business value to the application. The new provider object is created for every request made to Keycloak, as its lifespan is limited by the request execution.

Keycloak uses the concept of leveraging *Service Loader* for discovering all implementations of the *Spi*, *Provider*, and *ProviderFactory* interfaces. Basically, in order to use the custom providers and factories, they need to be registered in the application. It is done through the specification of class names of the providers and factories in files located in the *META-INF/services* folder.

4.2.2 Configuration

Keycloak nourishes the ability to configure all available SPIs through the dynamically assembled configuration properties. In that case, changing the particular value of a specific provider implementation is achievable. Enabling and disabling particular providers is also possible.[28]

Providers can be configured by using a specific configuration format such as follows:

$$spi\text{-}\langle spi\text{-}id\rangle\text{-}\langle provider\text{-}id\rangle\text{-}\langle property\rangle{=}\langle value\rangle \tag{4.1}$$

Where the explanation of these items is as follows:

- *<spi-id>* is the name of the SPI.
- *<provider-id>* is the identifier of the provider.
- *<property>* is the actual name of the configured property.

4.3 Authentication Flow

Authentication flow is a container for all authentications, screens, and actions that must happen during login, registration, and other Keycloak workflows. It is essentially a step-by-step process that users need to go through during authentication.

Keycloak consists of several authentication flow types, and part of them are described as follows:

- *Login flow* – outlines the necessary credentials a user needs to provide to gain access.
- *Registration flow* – dictates the required profile information a user must provide in order to be able to register and whether additional measures like reCAPTCHA are necessary to prevent bot activity.

- *Credential reset flow* – specifies the actions a user must take before being able to reset their password.

Authentication flow empowers administrators to craft tailored authentication processes for specific needs. By orchestrating a series of steps and actions, administrators can ensure secure authentication while accommodating diverse user requirements.[29]

Every authentication flow can have numerous inner sub-flows, which might contain another layer of sub-flows and actual authentication executions described in section 4.4. The modular nature of authentication flows enables administrators to mix and match components, facilitating adaptability and reusability.

4.4 Authentication Execution

Authentication executions are the foundational elements of authentication flows. These executions allow for the creation of comprehensive authentication flows, providing flexibility and control to address various scenarios effectively.

Ultimately, authentication executions serve as the cornerstone for building robust and user-centric authentication experiences within systems.

Authentication executions may play different roles in the context of particular authentication flows and provide the ability to control authentication flows in a more fine-grained way. [29]

Every authentication execution has a specified requirement type in its parent flow as follows:

1. **Required** – all *required* elements in the flow must be successfully sequentially executed. The flow terminates if a *required* element fails.

2. **Alternative** – a single element needs to execute successfully for the flow to be considered successful. If a flow includes *required* elements, they alone determine its success, rendering any *alternative* elements within it unnecessary.

3. **Disabled** – the element does not count to mark a flow as successful.

4. **Conditional** – (only set on sub-flows) if all executions evaluate as *true*, the *conditional* sub-flow acts as *required*, otherwise *disabled*.

4.5 Basic Authentication Flow

Authentication flows in Keycloak are very flexible and can address the unique needs of organizations. To illustrate the relationships between flows, sub-flows, and executions, Keycloak renders a graph with these elements, as shown in Figure 4.1.

The authentication flow contains several authentication flows and authentication executions. In the graph, every sub-flow is noted by a circle with the inner text *Start <auth-flow-name>*, and the end of the sub-flow by a circle with the inner text *End <auth-flow-name>*.

The flow contains on the first level 3 elements as *Cookie*, *Kerberos*, and flow *WebAuthn forms*. These elements are marked as alternatives, and if one of them succeeds, the authentication can proceed. When the *Cookie* and *Kerberos* executions are not fulfilled, the *WebAuthn forms* flow starts.

When a user provides their credentials via the Passwordless WebAuthn authenticator, the authentication is successful. Otherwise, a user needs to provide their password and then biometrics via the WebAuthn authenticator.

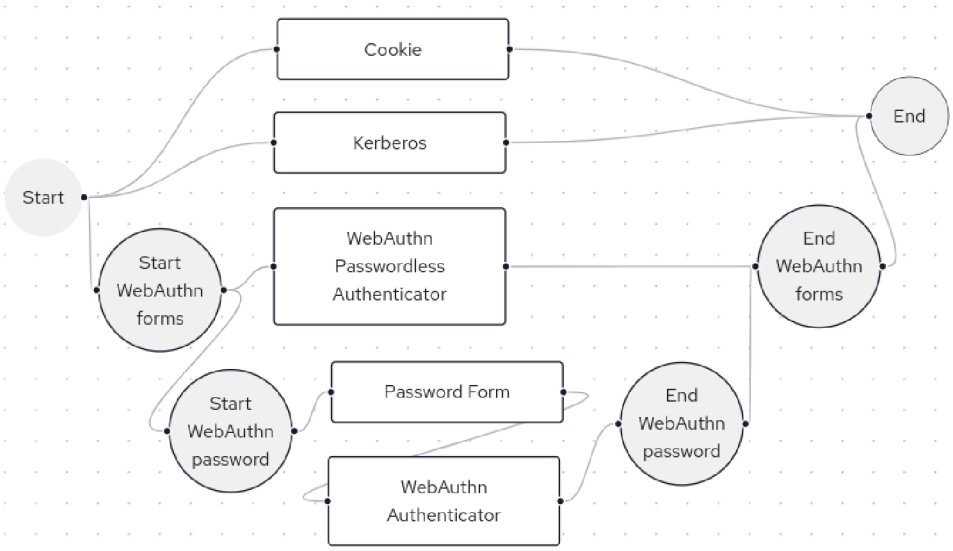

Figure 4.1: WebAuthn Flow.

5 Specification

The goal of this thesis is to enhance authentication processes in Keycloak. Adaptive authentication could promote Keycloak to a better place on the market by providing such a feature. The Keycloak community could benefit from it as well, as many other developers might be interested in the feature and be more engaged in the development.

The primary goal of this thesis is not to implement the whole solution and get it merged into the Keycloak repository. It is rather focused on providing an analysis of how it could be achieved, how it could work, and in what way it could be implemented. The output of individual items could be described as Proof of Concept (PoC).

PoC might be defined as a realization of a certain idea, method, or principle in order to demonstrate its feasibility or viability. It could also verify that some concept or theory has practical potential, so it might be very useful in this situation.[30]

The overall structure of the thesis might be divided into several points and described as follows:

1. **Extendability** – Provide suitable SPIs for the adaptive authentication components.

2. **Basic implementation** – PoC for a basic implementation of the adaptive authentication solution.

3. **Policy engine** – PoC for an approach leveraging authentication policies.

4. **Artificial intelligence** – PoC for an approach leveraging artificial intelligence.

5. **External integration** – PoC for an approach leveraging third-party services.

5.1 Extendability

One of the aims of the thesis is to provide reasonable extendability of the approach. Specifically, suitable SPIs should be created for the adaptive authentication components in order to have the ability to tailor them to the unique requirements of a user. Keycloak offers a very elegant way to extend components of the application without much hassle.

Users should be able to provide a custom approach for obtaining risk(contextual) factors data in an easy way. It means that users can create new or edit the default risk factor providers via the SPI concept described in Section 4.2. For example, obtaining the IP address of the device trying to authenticate is supplied from a proxy or load balancer outside of the application. Additionally, the evaluation of the risk factors should be flexible and customizable. Users or organizations extending the approach should be able to provide their own algorithms for the risk score evaluation.

5.2 Basic Implementation

The other objective pursued in this thesis is to provide a PoC for the basic implementation of the adaptive authentication solution. The approach should be able to be included in an authentication flow, gathering risk factors data, evaluating them, determining what level of risk the authentication engine is facing, and appropriately reacting to it. Users should be able to accommodate the solution into the authentication process and see the present outcome.

The basic implementation should contain an approach on how to assign certain authentication sub-flows, or authentication executions, to the decision branches based on the risk level. Users should be able to define multiple risk levels and customize ranges of risk scores associated with the levels.

Users might also be capable of setting certain decision weights to the data supplied by the risk factors mechanisms. This means that they can decide how much a particular risk factor should affect the overall risk score. It is also imperative to have the possibility to disable the evaluation of the risk score from particular risk factors.

5.3 Policy Engine

The next goal of the thesis is to create a PoC for authentication policy evaluations. Authentication policy should behave in such a manner, that comprehensive creation of conditions should be supported and adequately evaluated. Specifically, authentication policy can define the exact flow that would be executed when prior conditions are met.

For example, a particular authentication policy could require MFA when the device is located outside of the US. In that case, a user needs to specify the initial conditions as evaluating the location and specifying concrete action that will proceed.

The authentication policy might have the possibility of getting risk factors data and determining the action based on their values. The policy could also explicitly use the calculated risk score to decide the authentication attempt's continuity.

The authentication policy engine would gather and execute all the policies in a specific way. The engine could support the configuration of its behavior and manage data about the associated policies. Users should be able to replace the default authentication policy engine provider with a custom provider based on specific needs and requirements.

5.4 Artificial Intelligence

The other objective pursued in this thesis is to provide a PoC for leveraging artificial intelligence (AI) in the adaptive authentication approach. Nowadays, AI and machine learning are rapidly evolving with promising results, and the range of their applicability is more expansive than before. That approach might be very suitable even for adaptive authentication, as it might be used for calculating risk scores based on data from risk factors.

Data from risk factors might be analyzed and processed by AI mechanisms, which might lead to more educated decisions. The goal of leveraging artificial intelligence is to suggest an approach to what parts of the solution might benefit from using AI and run some experiments with AI engines.

5.5 External Integration

The last goal of the thesis is to provide a PoC for integrating third-party services into the current solution. Integration with external services, in some cases, might overlap with the previous goal in Section 5.4 related to AI. Specifically, AI engines are usually some sort of remote service with predefined API.

Integration with external services would be very beneficial for obtaining data from risk factors, which might be remote. However, one needs to be aware of possible issues with remote services and the overall authentication process execution. The credibility of remote sources may differ and needs to be adequately investigated to determine whether it is suitable to include in the risk score evaluation. Additionally, the latency for obtaining the data might be very high, and some sort of timeout should be set for gathering the risk factors data.

It leads to the specification of providing a particular risk factor data processing, as the retrieval of these data should be done asynchronously. As mentioned above, the timeout for each particular set risk factor should be set and respected during the evaluation. It may be configurable what to do in such situations – either to not include the risk factor in the evaluation or just decrease the weight of the risk factor.

6 Design

6.1 User Context

During the authentication process, the crucial part of a user successful authentication is to get information about multiple authentication factors, as described in Section 2.4. It is an essential part of security strengthening as the more information we get from the authentication attempt, the more we can react to it and evaluate it.

User context is a concept that represents a piece of information about the authentication attempt, such as current device attributes, information about the user, location, or behavior we detected. Even though the information about the device does not seem to reflect the term user context, it should be considered like it. User context represents the information about the user of the application, even when it is a service account, for instance.

During the authentication process, user contexts are always collected and grouped into a more comprehensive knowledge base. The proper user context management is a crucial part of the authentication, especially for the risk-based authentication mechanism.

The abstraction around fetching user contexts provides the ability to easily extend and process possible user context information and gives the freedom to obtain various types of contexts, even from remote locations.

The main advantage of using user contexts is to get the required information, regardless of the source of the information. It means that it does not really matter which approach was used for obtaining the information but that the information is available. It implies that administrators need to be aware of present user context providers/implementations in order to prevent any inaccuracies.

As shown in Figure 6.1, user context is represented as abstract class *UserContext*, where the generic is used. This means that every implementation of the *UserContext* abstract class can retrieve different types of data.

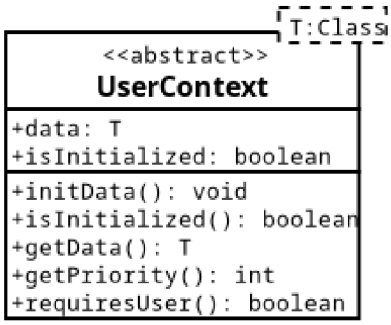

Figure 6.1: User Context Diagram.

Methods of the *UserContext* abstract class:

- *initData*() – a method that initiates retrieving the data.
- *isInitialized*() – a Boolean flag to determine whether the data has already been initialized.
- *getData*() – obtain the retrieved data.
- *getPriority*() – a priority for determining the order of retrieval of contexts of the same type. The higher the priority, the more preferred context it is.
- *requiresUser*() – a Boolean flag to determine whether the data can be obtained without the information about the authentication user.

6.1.1 Specific User Context

An excellent example of leveraging user context is the implementation of an IP address context. It provides the ability to obtain the IP address of the device in multiple ways. The IP address can be obtained either from the user agent attributes or from the HTTP headers *Forwarded* and *X-Forwarded-For* when a proxy is used.

As seen in Figure 6.2, the *IpAddressContext*[1] abstract class extends the *UserContext* abstract class and specifies that the retrieved data will

1. The naming convention assessed for the Keycloak project notes that the Pascal case used for class/interface names starts only with the first character uppercase, even with acronyms and abbreviations.

be of type *IPAddress*[2], which is a structure describing the IP address attributes.

The *DeviceIpAddressContext* class is an implementation of the *IpAddressContext* abstract class and obtains the IP address from the user agent attributes. The *HeaderIpAddressContext* class is also an implementation of the *IpAddressContext* abstract class and obtains the IP address from the HTTP headers when a proxy is used.

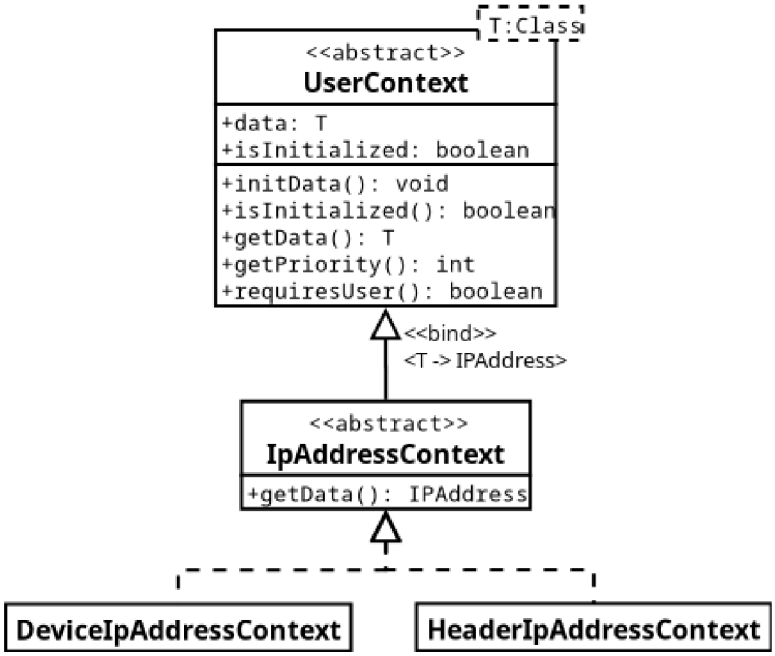

Figure 6.2: IP Address User Context Diagram.

2. A different naming convention as it describes an external third-party class.

6.1.2 User Context Condition

Improvement to the user context management is to have a general concept for evaluating certain conditions with regard to the user context. It means that we can leverage user context conditions in conditional authentication flows or in authentication policies, as described in 6.2.

There might be several possible operations made on the user context entities, and the developer of the additional user context should not reinvent the wheel around attributes of conditions. The additional conditions can be created very easily, as the general concept for evaluating them has already been assessed.

The abstract *UserContextConditionFactory* class implements the *VerifiableUserContextFactory* interface, which manages the *Operation* class described in Section 6.1.2.

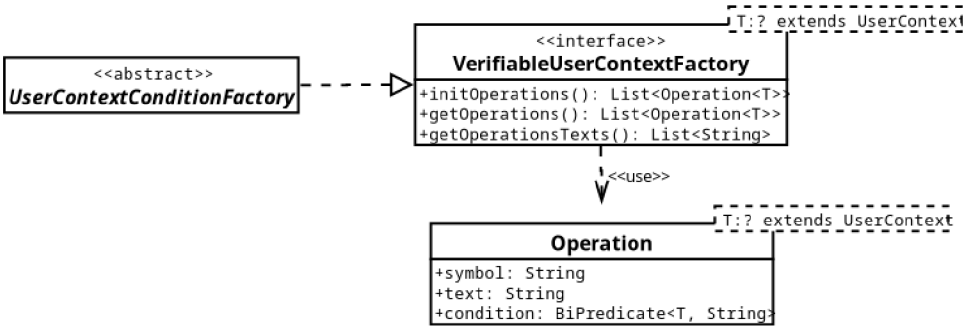

Figure 6.3: User Context Condition Diagram.

Operations

User context conditions take into account a list of operations that can be proceeded on the user context entity. The operation consists of several attributes, shown in Figure 6.4, such as follows:

- **Symbol** (*string*) – unique operation identifier used for identification of the operation when serialized data are received from remote locations. It must be unique per the user context condition.
- **Text** (*string*) – name of the operation shown to the end user. It can contain a localization key, which can then be mapped to a different language or directly the text in the specific language.
- **Condition** (*predicate*) – predicate that is applied to the specific user context. When the evaluation of the predicate is met, the operation is considered successful.

When all the predicates of specified operations are met, the whole condition is considered successful, and the authentication flow can proceed with the processing.

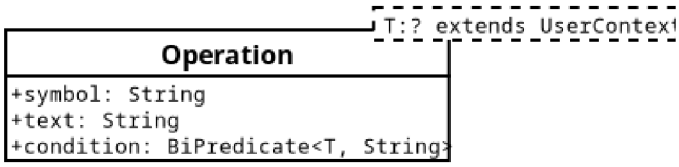

Figure 6.4: Operation Diagram.

Example

A suitable example of the user context condition might be a newly introduced condition for the IP address user context entity. During the evaluation of the condition, the first step is obtaining the IP address context, which contains the current IP address of the device that is trying to authenticate. Some specified operations that can be made on top of the IP address context are *is equal*, *is not equal*, or *is in range*.

The predicate of the *equal* operation is to directly match the IP address provided by the administrator. The opposite *not equal* operation just negatively matches the IP address, as it represents the state when the IP address is not the specified one. Administrators can also specify the *in range* operation, which provides the ability to check whether the IP address of the device is in the specified IP address range.

For instance, an administrator can specify that IP addresses from the range *222.0.0.0-224.0.0.0* are not allowed to be authenticated, as shown in Figure 6.5.

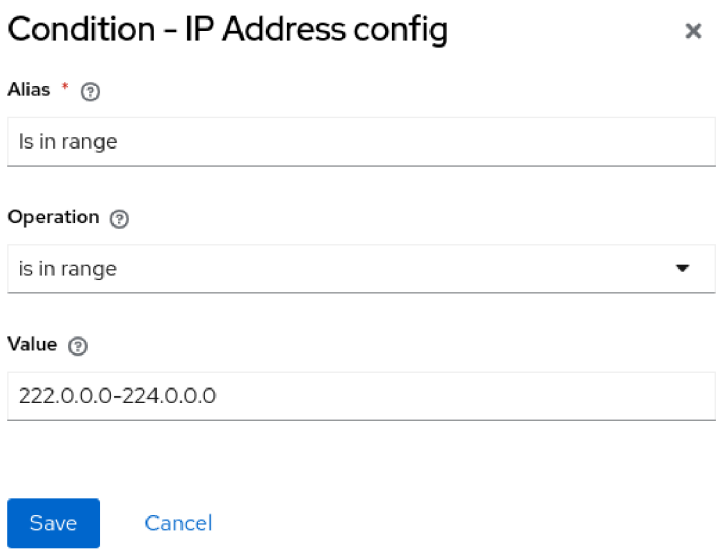

Figure 6.5: IP Address Context Condition.

6.2 Authentication Policies

Design of the authentication policies, or as described in the specification as a Policy engine, provides the ability to comply with requirements specified by the administrator. When users are trying to access the application, they need to meet certain conditions in order to achieve a successful login. There may be some company or federal restrictions to particular services that need to be conformed to.

For instance, there might be some requirements to deny users access to the application from a specific country or specific types of devices. It needs to be determined in which authentication process phases these policies should be evaluated. Policies might be required to comply with some preconditions in order to be able to evaluate them. It means that the environment context needs to be aligned with the policy requirement, such as gathering information about the user attempting to log in.

A particular hierarchy in policy priorities needs to be assessed as well in order to evaluate specific policies before others. It provides the capability to create a more complex and comprehensive evaluation pipeline for authentication policies.

Every authentication policy consists of several conditions and actions. All conditions must be met in order to execute specific actions. If some of these conditions are not evaluated to be true, the whole authentication policy is skipped, and the following policy in the hierarchy structure is evaluated.

Authentication policies can be specified interactively via the administrator console or *HTTP REST API*. All of them are stored in persistent storage, so every change of the structure is visible even after the re-initialization of the application. Specific operations can be executed on the policy entities, such as obtaining a specific authentication policy, creating a new one, or updating or removing the existing one. Moreover, the priority to achieve the required hierarchy structure can be defined as well.

6.2.1 Authentication Flow

Authentication policies are tightly coupled to the authentication flows described in Section 4.3. Authentication policies are a specific subset of authentication flows, as these policies are basically just conditional authentication flows.

Conditional flows provide the majority of the required functionality to meet requirements for authentication policies or the policy engine. It means that specific conditions are evaluated, and if all pass, specific actions are executed.

One of the major issues around authentication policies is manageability. Administrators should not add a higher number of conditional flows to their existing authentication flows, as these concepts should be separated in this manner.

Authentication policies extend the capabilities around the authentication flow process as they can be defined for the whole realm, not even for a specific flow. To be more specific, it gives the possibility to share policies across different flows, which is the main difference between conditional flows and authentication policies. It provides more fine-grained authentication policy management and a better authentication settings approach.

6.2.2 Authentication Policy Authenticator

In order to properly evaluate all authentication policies specified for the realm without the need to comprehensively expand the authentication flows, a specific authenticator was introduced. It provides the ability to have a more fine-grained approach for these evaluations, as the administrator can specify in which phases of the authentication flow these policies will be processed.

The authenticator works as a configurable placeholder for these policies, as it can contain multiple configuration properties to asses the overall policies processing. It might be possible to hide the authenticator from the flows completely, but the explicit settings provide a more deterministic approach.

The default authenticator configuration contains only a Boolean flag to determine whether the user information is required for the evaluations or not.

When the authentication flow processing is in the phase of evaluating the authenticator, multiple steps are executed:

1. **Obtain** – Obtain all authentication policies.
2. **Filter** – Filter policies based on the authenticator configuration.
3. **Process** – Process all authentication policies.
4. **Return** – Return to the parent authentication flow and continue with processing other steps.

When all conditions are met for an arbitrary policy, and the action explicitly allows access to the user, the authenticator itself is evaluated as successful without any other policy evaluations.

The same applies to the case when the action explicitly denies access to the user, the authenticator is evaluated as false, and the whole authentication flow processing is stopped.

Otherwise, when the action is not terminal, the other policies are evaluated as usual.

6.2.3 Policy Evaluation Example

For a better representation of the authentication policy evaluation, a simple example with the described configuration might be considered. The specification of a particular authentication policy is done through the administrator console in the Authentication section and Authentication Policies tab.

The following simple example of the authentication policy, in Figure 6.6, has one condition and one action. If the first condition is met, the underlying action is executed.

The policy contains a condition based on the current browser properties, which does not allow a specific browser vendor – Mozilla Firefox, in this case. When the condition is met, access to the application is denied to the user. This means that authentication requests can only be made via a browser other than Mozilla Firefox.

The condition configuration attributes differ based on the used condition provider/implementation. Every condition provider specifies a set of configuration properties for condition settings.

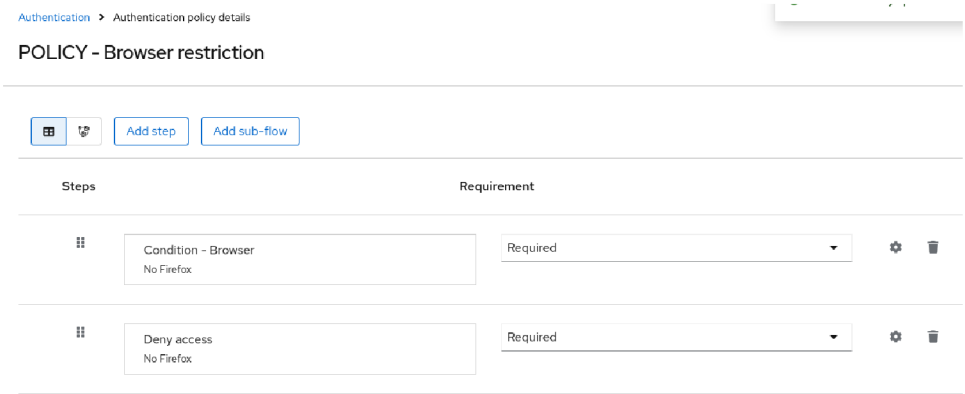

Figure 6.6: Authentication Policy Browser Restriction.

The browser condition, shown in Figure 6.7, includes properties:

- **Alias** – name of the condition.
- **Operation** – list of possible operations.
- **Browser** – multi valued list of browsers.

In this case, the condition is met only when the browser vendor is Firefox:

The action configuration also differs based on the provider of the action. For the Deny access provider, visible in Figure 6.8, the only additional property is *Error Message*. It represents an error message that is shown to the user when the access is denied.

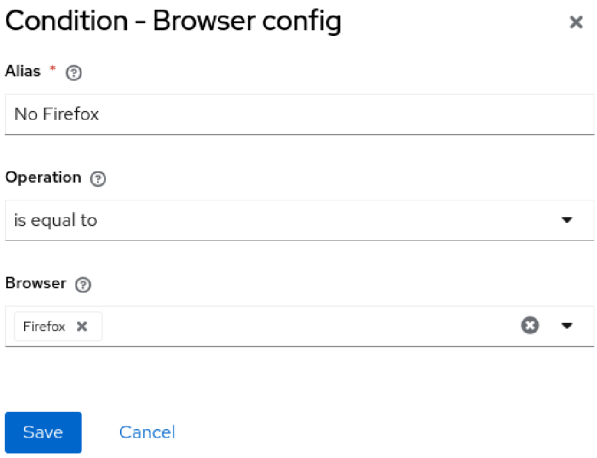

Figure 6.7: Browser Restriction Condition.

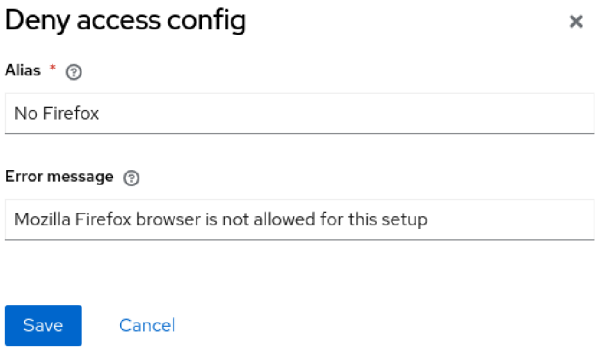

Figure 6.8: Browser Restriction Deny Access.

6.3 Risk-based Authentication

The crucial part of adaptive authentication is focused on risk-based authentication. Risk-based authentication is a concept that assesses a risk level and then adjusts authentication requirements accordingly. The risk in the risk-based authentication represents a probability that the authentication attempt is fraudulent.

The concept is based on this core sequence:

1. **User Context Collection** – Gather all available user contexts for the following processing.
2. **Risk Scoring** – Assess the risk score for all the user contexts.
3. **Adaptive Adjustment** – Adaptively adjust authentication flow based on the overall risk score.

The collection of user contexts is an essential part of the whole risk-based authentication concept – as more information is present, the more informed decisions can be made. There is a defined timeout for obtaining the user context, as it may take a vast majority of time, and administrators are able to adjust it to their needs.

Obtaining user context from remote locations or services may be time-consuming, and the overall user experience around the authentication process would be worse. When the timeout is exceeded, the user context is not taken into account for the risk evaluation.

When the user context collection is successful, the risk is evaluated for every user context. The exact approach and how it is achieved is described in 6.3.1.

When all these evaluations are done, the overall risk for the authentication attempt is calculated with the usage of various algorithms described in 6.3.2.

After calculating the overall risk, the risk level is determined based on the risk score, described in 6.3.3.

6.3.1 Risk Evaluators

Risk evaluators play a crucial role in risk-based authentication, as their primary purpose is evaluating risk for specific user contexts in a fine-grained, modularized way.

The evaluation itself might be a challenging area to handle, as there are no guidelines on how exactly it should be done. It requires specific expertise and multiple agreements on how exactly the risk should be calculated for each individual user context.

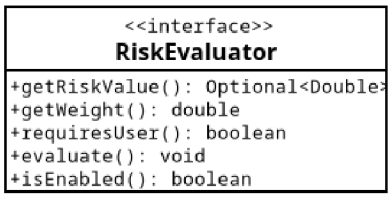

Figure 6.9: Risk Evaluator Diagram.

Attributes

The main risk evaluator attributes:

- *risk* – evaluated risk in range (0,1>.
- *weight* – weight of the evaluated risk in range (0,1> (default 0.5).
- *requiresUser* – flag determining whether the evaluation needs information about the user trying to authenticate.
- *enabled* – flag determining whether the evaluator risk should be calculated and taken into consideration for the overall risk scoring system.

Attributes *risk* and *weight* contribute to the overall risk scoring system in a magnificent way. The purpose of the *weight* attribute is to handle every risk evaluation with different weights. To be more specific, some of these evaluators can contribute more to the overall risk score than others. It provides the option to rely on evaluators where the evaluation confidence is higher, and it should influence

the overall risk more than evaluators with lower confidence in their evaluations.

Attribute *requiresUser* is a Boolean flag used to determine the accurate phase of the evaluation. Some evaluators need to be already aware of the attempting user, as they need the necessary information about the user. This means that risk evaluation itself needs to be done after obtaining the essential user information.

Attribute *enabled* is a Boolean flag used to check dynamically whether the evaluator should be turned on. Administrators can decide which evaluators are taken into account for the processing.

Scope

The main risk evaluator characteristic is related to the scope of the user contexts used for these evaluations.

Risk evaluators can be determined as:

- **Basic** – operates only on one user context.
- **Contextual** – operates on more user context and considers the whole context around obtained specific user contexts.

The implementation of a particular risk evaluator should be focused only on one information unit. It means that in order to provide fine-grained manageable risk evaluations, the resulting risk should be related only to one user context or component. In that case, the most suitable approach is to use basic risk evaluators, but sometimes, to properly assess the risk score, a broader context is necessary.

For instance, the risk evaluator handling the calculation of risk for the IP address should be related only to the IP addresses and nothing else. It may be a contextual risk evaluator, as it needs to obtain device user context, IP address context, or potentially information about proxy usage. It should not take into consideration a wider context out of its scope as that is not the intentional usage of the component.

For evaluating the overall risk score and considering the wider context, the risk engine component is provided and described in 6.3.2.

Configuration

The key concept around risk evaluators is their configuration for better manageability and user experience. Administrators have the ability to assess the main attributes of evaluators to their needs, such as tweaking the *weight* or *enabled* attribute without the necessity to change the risk evaluator implementation.

In the *Risk-based policies* section of the administrator console, properties for value amendment are automatically generated. When a developer creates another risk evaluator, the configuration for the basic configurable attributes is generated in the administrator console.

As shown in Figure 6.10, the risk evaluators configuration contains multiple similar configuration fields grouped together and noted via the colored rectangles.

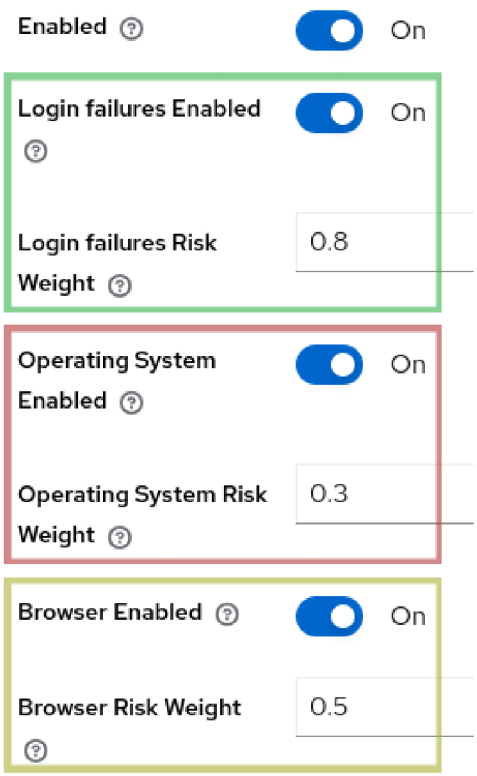

Figure 6.10: Risk Evaluators Configuration.

6.3.2 Risk Engine

The risk engine takes care of evaluating the overall risk score based on scores obtained from the risk evaluators. One of the primary purposes of the risk engine is to manage the underlying evaluators, such as processing their lifecycle phases and contextually evaluating the overall score.

Each implementation can provide its own approach to send a signal to evaluators to start evaluating risk scores or specify a different algorithm for the overall risk score.

The default risk engine provides a retry mechanism when the risk evaluator does not return any result. Failure may occur when obtaining the user context or evaluating the risk score. It is usually a problem of leveraging remote services, as the process is more comprehensive and includes several additional potential points of failure. When the processing fails, the risk engine requests to retry the evaluation again.

The default risk engine also introduces a timeout mechanism as the delivery time is guaranteed and upper bounded. Expressly, it is guaranteed that processing the risk for individual evaluators does not exceed the defined timeout and does not allocate the processor timeout with the following application unresponsiveness.

The overall risk score is evaluated continuously based on the phase of authentication. During the authentication flow, the risk may be evaluated multiple times, as the requirements for the risk evaluators and risk engine context may differ. The fundamental phases of risk engine evaluation are *NO_USER* and *REQUIRES_USER*.

The *NO_USER* phase represents the phase before determining who is trying to access the application. It means only information about the application stack is present, such as information about the device and network.

The *REQUIRES_USER* phase represents the phase when the user is already known during the authentication process.

Default Algorithm

The default algorithm for calculating the overall risk score is weighted arithmetic mean, which provides the necessary characteristic for the risk evaluation.

The weighted arithmetic mean shares similarities with the standard arithmetic mean, the typical average calculation method. However, unlike the arithmetic mean, where each data point holds equal weight in determining the final average, the weighted arithmetic mean assigns varying weights to different data points.

This is an exact concept suitable for leveraging risk evaluators and aggregating their results. When all weights are uniform, the weighted mean converges to the arithmetic mean.

The formula for the weighted arithmetic mean:

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i} \tag{6.1}$$

The $\bar{x}$ represents the overall risk score, x_i represents the individual risk evaluator score, and w_i represents the corresponding weight for each risk evaluator.

The overall risk of the whole authentication process is evaluated as the maximum risk score of each individual phase. It means when the overall score for phase *NO_USER* is *0.7*, and the score for phase *REQUIRES_USER* is *0.6*, the overall risk score for the whole process is *0.7*.

Design

The risk engine is presented as *RiskEngine* interface with several methods shown in Figure 6.11. The interface extends *ConfigurableRequirements* interface for achieving dynamic check whether the authentication user is required during the risk engine processing. There is also a weak relationship between *RiskEngine* and *RiskEvaluator* interface, as *RiskEngine* contains a collection of the risk evaluators. Therefore, the aggregation relationship is used in this case.

Risk engine methods:

- *evaluateRisk*() – method that initiates the evaluation of the collected risk.
- *getRisk*() – retrieve the evaluated risk.
- *getRiskEvaluators*() – retrieve all risk evaluators that contribute to the risk calculations.

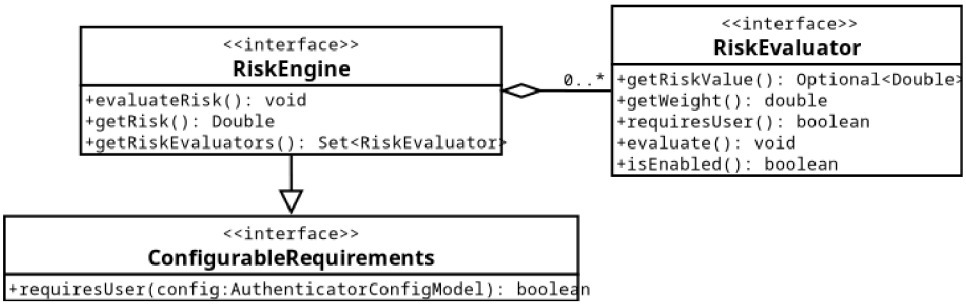

Figure 6.11: Risk Engine Diagram.

Configuration

It is essential for administrators to have the capability to set reasonable values for the timeout and number of retries for the risk engine. Besides configuring the overall state of risk-based functionality, it may be done through the administrator console, as shown in Figure 6.12. The default number of retries is *3*. The default timeout for processing the risk score is *1500 ms*.

Stored Risk Provider

As the overall risk score is evaluated in several phases, the results for every phase need to be available in some manner. The most reasonable approach would be storing the data in a particular place for later processing.

The lifespan of these data should not exceed the lifespan of the authentication request. However, when some action is required from

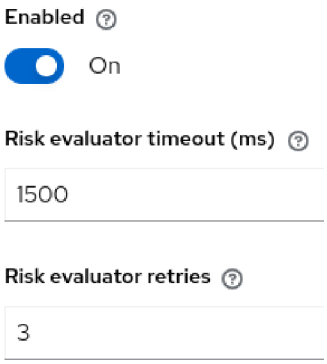

Figure 6.12: Risk Engine Configuration.

the user, such as providing a password, the HTTP and authentication requests are different.

The start for the authentication request might be considered at the beginning of processing the authentication flow when the Keycloak authentication session is created. Therefore, leveraging authentication session notes seems to be the most reasonable way to store the risk.

However, these data might be stored in different stores, such as databases or cache, and the developer should have the ability to extend the functionality. Thus, the *SPI* for the stored risk is introduced. The provider for storing the risk is represented by *StoredRiskProvider*, shown in Figure 6.13.

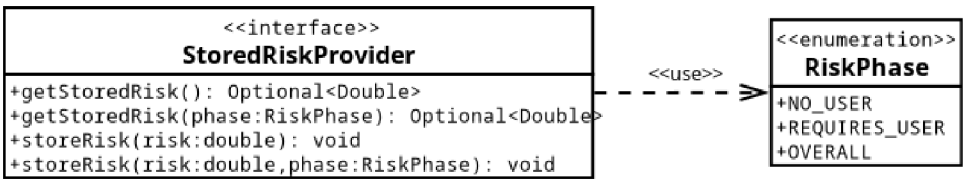

Figure 6.13: Stored Risk Provider Diagram.

Stored risk provider methods:

- *getStoredRisk*() – retrieve the overall stored risk score.
- *getStoredRisk*(*phase*) – retrieve the risk score for a specific risk phase.
- *storeRisk*(*risk*) – store overall risk score.
- *storeRisk*(*risk, phase*) – store risk score for a specific risk phase.

Risk phase values representing the phase of the evaluated risk score are described in detail in the general subsection of 6.3.2.

Risk Engine Authenticator

The primary purpose of the risk engine authenticator is to explicitly trigger the risk engine evaluations for specific risk processing phases. It is the standard Keycloak authenticator so that it can be included in a specific place in an authentication flow. The intention behind introducing the authenticator is to have better control over the risk processing lifecycle without the need to touch the Keycloak codebase out of the Keycloak extension.

It might be possible to hide the authenticator for administrators, as it would be evaluated once the risk processing phase met some specific condition that checks the ability to process it. The authenticator configuration might be seen in Figure 6.14.

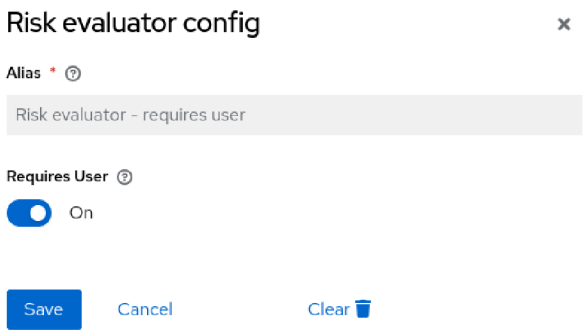

Figure 6.14: Risk Engine Authenticator Configuration.

6.3.3 Risk Levels

When the risk engine calculates the overall risk score for the authentication request, the application needs to react to it in some manner.

The primary purpose of the risk levels is to divide risk score ranges into specific levels. The risk level is basically an entity with a specified lower and upper bound of risk score shown in Figure 6.15.

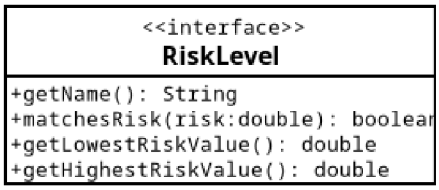

Figure 6.15: Risk Level Diagram.

Risk level methods:

- *getName*() – retrieve the name of the level.
- *matchesRisk*(*risk*) – check whether a specific risk is associated to a risk level.
- *getLowestRiskValue*() – retrieve the lower risk bound.
- *getHighestRiskValue*() – retrieve the upper risk bound.

As the risk score might consider dynamic evaluations, which can provide slightly different fluctuating results, relying on specific risk score values is impossible. That means a reasonable granularity needs to be assessed to manage the overall risk score – several levels.

There are two default distinct risk level categories providing different granularity of risk levels – *simple* and *advanced*. Developers can provide their custom risk level category by implementing the *RiskLevelsSpi*.

The *simple* risk level category consists of three risk levels:

- **Low** – Risk score in range (0, 0.33>.
- **Medium** – Risk score in range (0.33, 0.66>.
- **High** – Risk score in range (0.66, 1>.

The *advanced* risk level category consists of five risk levels:

- **Low** – Risk score in range (0, 0.2>.
- **Mild** – Risk score in range (0.2, 0.4>.
- **Medium** – Risk score in range (0.4, 0.6>.
- **Moderate** – Risk score in range (0.6, 0.8>.
- **High** – Risk score in range (0.8, 1>.

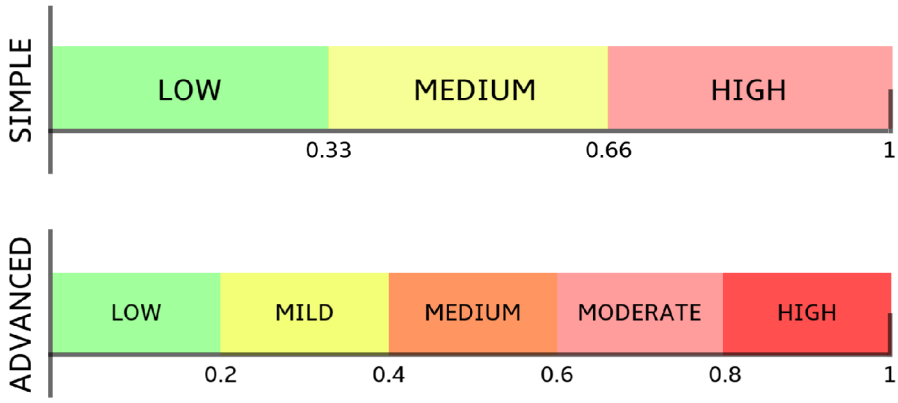

Figure 6.16: Default Risk Levels.

Risk Level Conditions

In order to accommodate the risk levels mechanism into the authentication flow, expressly conditional flows or authentication policies, the risk level condition is introduced. The primary purpose of the condition is to process conditional flow or authentication policy when its predicate is evaluated as *true*.

In this case, the predicate obtains the stored risk from the risk engine and, based on the specified risk level category, determines whether the risk is included in the risk level category. If the evaluated risk score is in the range of the specified category, the condition is evaluated as *true*, and the underlying flow is processed.

The risk level condition contains a configuration list box with values defined for the particular risk level provider, as shown in Figure 6.17.

Figure 6.17: Simple Risk Level Configuration.

Risk Level Flow Example

A proper example of leveraging the risk levels in the conditional flows or authentication policies might be seen in Figure 6.18 below. The risk level flow is a sequence of conditional flows consisting of risk level conditions and actions that should be executed.

For better visibility, the conditional flows are marked by different colors. In the case shown in Figure 6.18, consider the *simple* risk level category. When the evaluated risk is:

- **Low** – the access to the application is granted without any other action.
- **Medium** – the second factor is required – using OTP.
- **High** – the access is denied.

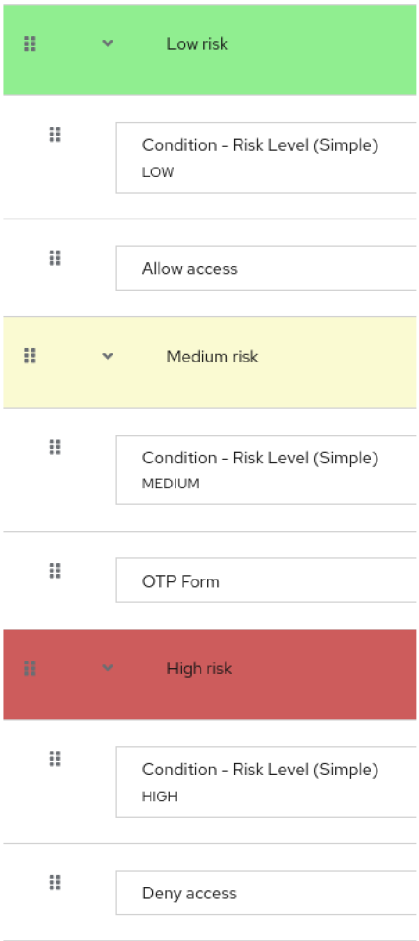

Figure 6.18: Simple Risk Level Flow.

6.4 Artificial Intelligence

Artificial intelligence (AI) and *machine learning* (ML) are rapidly evolving with promising results, and the range of their applicability is more expansive than before. As mentioned previously, evaluating risk scores for user contexts is a complex discipline with many boundaries and required assessments. For more details and specific requirements for the AI approach, see Section 5.4.

With the help of AI, the problem can be simplified as it could potentially evaluate the risk score more accurately based on the overall context. However, the evaluation of the risk by AI might bring its own risk of unpredictability when the machine learning model is not mature in a higher measure.

The model needs to determine a fraudulent request with a higher probability and gain knowledge from the ongoing traffic to the application. Modeling a custom AI model is a challenging topic and out of the scope of this thesis due to its complexity.

The design for integrating AI services provides extendable API for leveraging multiple AI engines and being vendor-agnostic as much as possible. The most suitable place for leveraging AI is evaluating risk scores for user contexts. Due to the unpredictability and non-deterministic behavior of AI evaluations, the design takes care only of specific risk evaluations.

To be more specific, the design does not suggest implementing a risk engine based on AI but rather creating a hybrid solution with static evaluations in contribution to the dynamic AI evaluations.

The design leverages *Natural Language Processing* (NLP) systems, as the request on the service contains commands queried in a natural human language. This means the system needs to understand the task, the context, and the specific user message.

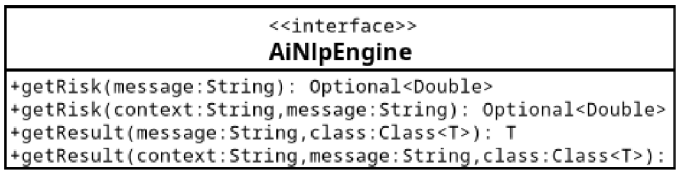

Figure 6.19: AI NLP Engine Diagram.

For the integration of the NLP systems, the *AiNlpEngine* interface has been introduced as it provides a unified API for AI NLP engines. Every implementation of the engine needs to implement these methods:

- *getRisk*(*message*) – retrieve evaluated risk score queried by user *message*.
- *getRisk*(*context, message*) – retrieve evaluated risk score queried by user *message* and a specific *context*.
- *getResult*(*message, class*) – retrieve serialized data of type *class* with the user *message* and the default *context*.
- *getResult*(*context, message, class*) – retrieve serialized data of type *class* with the user *message* and a specific *context*.

Specific risk evaluators leveraging AI use for communication with the AI service a specific *AiNlpEngine* implementation in order to obtain the required risk score evaluated by the service, as shown in Figure 6.20.

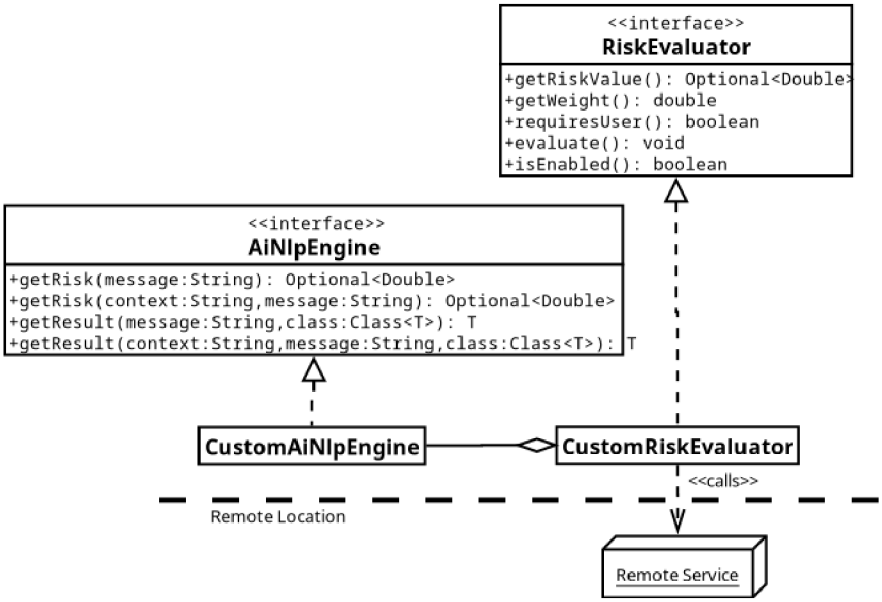

Figure 6.20: Risk Evaluator and AI NLP Engine Diagram.

6.5 External Integration

External integration with third-party services can be smoothly adjusted due to the design of the components for adaptive authentication. For more details and specific requirements for the external integration, see Section 5.5.

External integration is also related to the previous Section 6.4, as the AI services can be queried due to the external integration capabilities. It is possible to attach a remote service to the adaptive authentication process managed by the risk engine, as described in Section 6.3.2.

The design of the remote user context is shown on the Diagram 6.21, where the information about device location is fetched from the remote third-party service. The *LocationContext* specifies the format of data it works with and the custom implementation needs to provide it. Therefore, the application still seamlessly works with the location data despite the implementation of retrieving the location information.

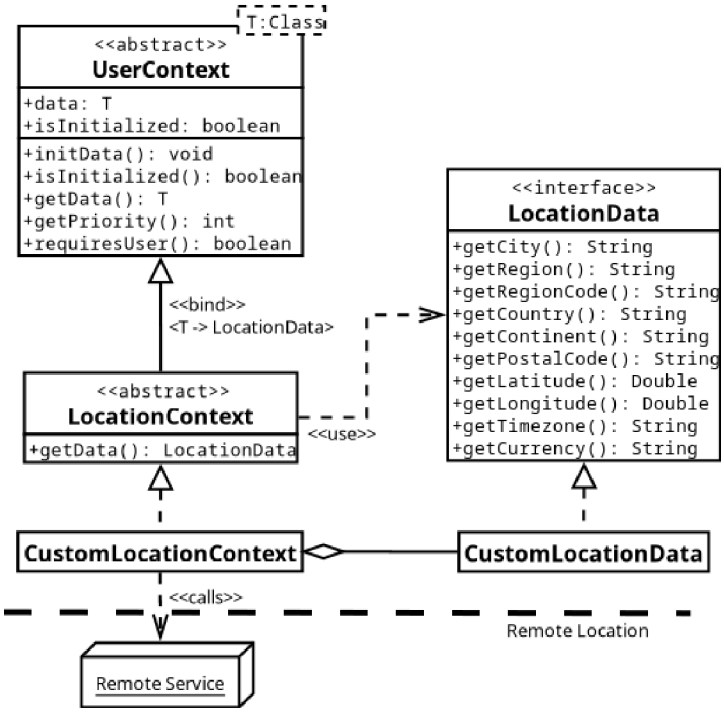

Figure 6.21: Remote Location Context.

7 Implementation

7.1 Authentication Policy

With regard to the design of the authentication policies, conditional flows are used to fulfill the requirements for authentication policies. The main difference is the additional lifecycle management provided for them with particular validations. Reusing existing components is beneficial in this case as the implementation is simplified without introducing any other similar components.

In order to differentiate authentication flow and authentication policy, the parent authentication flow *Authentication policies - PARENT* is provided to contain only authentication policies. Moreover, every policy has the prefix *"POLICY - "* for its alias to mark it as the authentication policy. The validation and management for authentication policies are done through the specific *Data Access Object* (DAO), shown in Section 7.1.1.

As the authentication policies need to be managed from the outside world, the *REST API* has been introduced with a detailed description in Section 7.1.2. The administrator console, as a client application, also uses the Keycloak REST API to manage all entities.

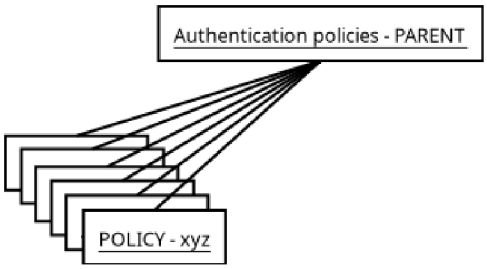

Figure 7.1: Authentication Policies Parent.

7.1.1 Data Access Object

Data Access Object (DAO) pattern in software engineering abstracts database operations, shielding application logic from database sophistication and adhering to the single responsibility principle.[31]

It is used to manage authentication policies as part of the separation from the authentication flows. It provides necessary validation above the database layer and provides the functionality to properly mark conditional flow as an authentication policy.

The signature of operations for the authentication policy DAO is declared via the *AuthnPolicyProvider* as shown in Figure 7.2.

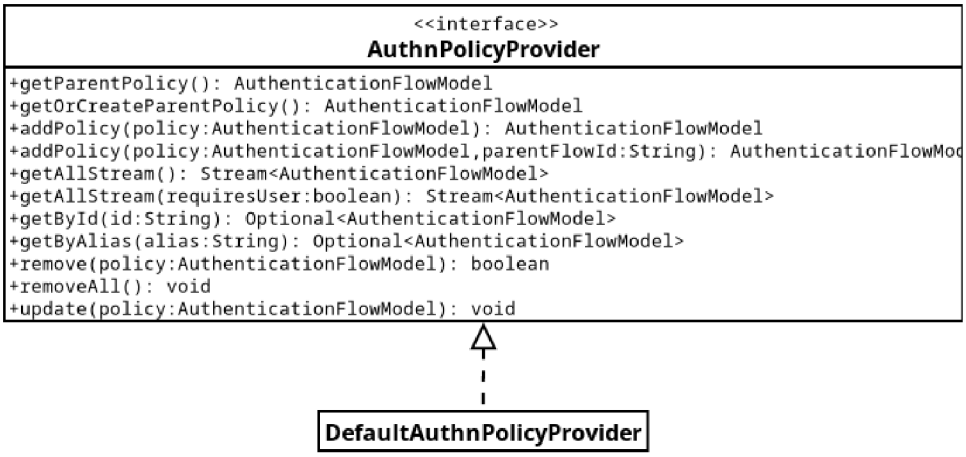

Figure 7.2: Authentication Policy Provider.

The interface *AuthnPolicyProvider* consists of the basic CRUD operations that can be made on authentication policies, such as:

- *getParentPolicy*() – retrieve the parent *Authentication policies - PARENT* flow.
- *getOrCreateParentPolicy*() – retrieve, or create if it does not exist, the parent *Authentication policies - PARENT* flow.
- *addPolicy*(*policy*) – add a new policy *policy* with the default parent flow.
- *addPolicy*(*policy, parentFlowId*) – add a new policy *policy* with the parent flow *parentFlowId*.
- *getAllStream*() – retrieve a stream of all authentication policies.
- *getAllStream*(*requiresUser*) – retrieve a stream of all authentication policies with the execution phase equal to *requiresUser* attribute.
- *getById*(*id*) – retrieve an authentication policy found by provided ID via the *id* attribute.
- *getByAlias*(*alias*) – retrieve an authentication policy found by provided alias via the *alias* attribute.
- *remove*(*policy*) – remove a policy defined in the *policy* attribute.
- *removeAll*() – remove all authentication policies.
- *update*(*policy*) – update an authentication policy provided via the *policy* attribute.

Add Authentication Policy

One of the most interesting examples of implementing methods of the *AuthnPolicyProvider* interface is the *addPolicy*() method shown in Figure 7.3. That method is responsible for creating a new authentication policy based on the parameters.

```
@Override
public AuthenticationFlowModel addPolicy(AuthenticationFlowModel policy, String parentFlowId) {
    if (StringUtil.isBlank(policy.getAlias()))
        throw new IllegalArgumentException("Cannot create an authentication policy with an empty alias");

    if (!policy.getAlias().startsWith(POLICY_PREFIX)) {
        policy.setAlias(POLICY_PREFIX + policy.getAlias());
    }

    var flow : AuthenticationFlowModel = realm.addAuthenticationFlow(policy);

    var execution = new AuthenticationExecutionModel();
    execution.setParentFlow(parentFlowId);
    execution.setRequirement(AuthenticationExecutionModel.Requirement.CONDITIONAL);
    execution.setPriority(0);
    execution.setFlowId(flow.getId());
    execution.setAuthenticatorFlow(true);

    realm.addAuthenticatorExecution(execution);

    return flow;
}
```

Figure 7.3: Add Policy Method.

First of all, the alias must not be empty. Otherwise, the exception is thrown. If the alias does not start with the prefix *"POLICY - "*, it is automatically added to the alias. The last part of the execution is to create a conditional flow where the parent flow is the common parent authentication flow for all authentication policies. The newly created authentication policy is returned.

Retrieve Authentication Policies

Another method worth mentioning is the *getAllStream*(*requiresUser*), shown in Figure 7.4. The method returns all authentication policies that match the *requiresUser* Boolean attribute. Specifically, it iterates over all policies, and if there is some requirement to require a user, it automatically includes it in the final stream. It does not consider several layers of authentication policies hierarchy. If the execution authenticator implements the *ConfigurableRequirements* interface, the condition of whether it requires a user is evaluated dynamically.

```
@Override
public Stream<AuthenticationFlowModel> getAllStream(boolean requiresUser) {
    Predicate<Stream<Boolean>> OPERATION = requiresUser ?
            s -> s.anyMatch(f -> f) :
            s -> s.noneMatch(f -> f);

    Predicate<AuthenticationFlowModel> REQUIRES_USER = f -> OPERATION.test(
            getAllAuthenticationExecutionsStream(f.getId()).map(g -> {
                var authenticator : Authenticator = getAuthenticator(session, g.getAuthenticator());
                if (authenticator instanceof ConfigurableRequirements configurable) {
                    return configurable.requiresUser(realm.getAuthenticatorConfigById(g.getAuthenticatorConfig()));
                } else {
                    return authenticator.requiresUser();
                }
            }));

    return getAllStream().filter(REQUIRES_USER);
}
```

Figure 7.4: Get All Authentication Policies.

The helper method *getAllAuthenticationExecutionsStream* shown in Figure 7.5 returns all executions included in the authentication policy with the usage of recursion.

```
private Stream<AuthenticationExecutionModel> getAllAuthenticationExecutionsStream(String flowId) {
    return realm.getAuthenticationExecutionsStream(flowId).flatMap(g -> {
        if (g.isAuthenticatorFlow()) {
            return getAllAuthenticationExecutionsStream(g.getFlowId());
        } else {
            return Stream.of(g);
        }
    });
}
```

Figure 7.5: Get All Authentication Executions.

7.1.2 REST API

A REST API (also known as RESTful API) is an API that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for *REpresentational State Transfer*.[32]

A client accessing the REST API through the parent endpoint */authn-policies* defined for the *AuthnPoliciesResource* request handler class. When an HTTP request is sent to the endpoint, it is handled by the class, which directs it to the appropriate method based on the HTTP method of the request, shown in Figure 7.6.

The request handler classes work explicitly with the authentication policy DAO, described in Section 7.1.1. It provides the isolation of storage operations and user interaction and complies with the Single-responsibility principle.[33]

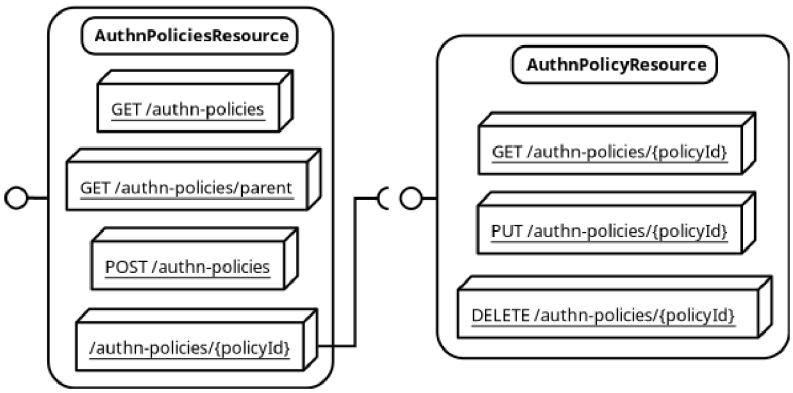

Figure 7.6: Authentication Policies REST API.

Possible operations on the *AuthnPoliciesResource* request handler class with the additional redirection:

- *GET /authn-policies* – retrieve all authentication policies in JSON format.
- *GET /authn-policies/parent* – retrieve the parent authentication policy in JSON format.
- *POST /authn-policies* – create a new authentication policy with attributes defined as JSON in the request body.

- */authn-policies/policyId* – Redirects requests to the *AuthnPolicyResource* request handler class.

Possible operations on the *AuthnPolicyResource* request handler class:

- *GET /authn-policies/{policyId}* – retrieve an authentication policy with ID *policyId* in JSON format.
- *PUT /authn-policies/{policyId}* – update an existing authentication policy with ID *policyId* with attributes defined as JSON in the request body.
- *DELETE /authn-policies/{policyId}* – delete an existing authentication policy with ID *policyId*.

The exact implementation of a specific part of the *AuthnPoliciesResource* class is shown in Figure 7.7. In the constructor, all necessary initializations are done, and the authentication policy DAO is injected. When the HTTP request with the *GET* request method is sent to the endpoint, the *getPolicies()* method returns the set of policies in a serialized JSON format. The abstract model representation is converted to the *Data Transfer Object* (DTO) and returned to the client.

```
private final KeycloakSession session;
private final RealmModel realm;
private final AuthnPolicyProvider provider;

public AuthnPoliciesResource(KeycloakSession session) {
    this.session = session;
    this.realm = session.getContext().getRealm();
    this.provider = session.getProvider(AuthnPolicyProvider.class);
}

@GET
@Produces(MediaType.APPLICATION_JSON)
public Set<AuthenticationFlowRepresentation> getPolicies() {
    return provider.getAllStream()
            .map(f -> ModelToRepresentation.toRepresentation(session, realm, f))
            .collect(Collectors.toSet());
}
```

Figure 7.7: Authentication Policies REST API Implementation.

7.2 User Context

The first step for implementing the *UserContext<T>* abstract class is to bind the generic type to the required one, either through extending the abstract class or via a direct reference from the implementation class.

For demonstration purposes, consider a specific user context *IpProxyContext*, which handles all IP addresses present in the HTTP *Forwarded*, and *X-Forwarded-For* headers used when a proxy is part of the deployment.

The specific *IpProxyContext* was created, shown in Figure 7.8, in order to bind the type of the user context with a possible extension of it.

```
public abstract class IpProxyContext extends UserContext<Set<IPAddress>> {
}
```

Figure 7.8: Extend *UserContext* Interface.

The implementation of the *IpProxyContext*, shown in Figure 7.9, consists of the implementation of the *initData*() method to fulfill the requirements of the abstract class.

With the help of the *KeycloakSession*, the data for a particular context is processed only once per request, so when different risk evaluators need the context, it is not evaluated multiple times.

However, the initialization of the data does not have to be successful. In that case, the *isInitialized* Boolean flag is *false*, and the user contexts processing unit may repeat the initialization based on the information.

During the initialization of the data, the HTTP headers of the current request are parsed. IP addresses included in the headers *Forwarded* and *X-Forwarded-For* are concatenated together. When there is valid data, the Boolean flag *isInitialized* is set to *true*, which notes the initialization is done correctly and data is available.

```
public class ProxyIpAddressContext extends IpProxyContext {
    private final KeycloakSession session;

    public ProxyIpAddressContext(KeycloakSession session) {
        this.session = session;
    }

    @Override
    public void initData() {
        var result : Optional<Set<...>> = Optional.ofNullable(session.getContext()) Optional<KeycloakContext>
                .map(KeycloakContext::getRequestHeaders) Optional<HttpHeaders>
                .map(headers -> Stream.concat(
                        getIpAddressFromHeader(headers, headerName: "Forwarded"),
                        getIpAddressFromHeader(headers, headerName: "X-Forwarded-For"))
                ) Optional<Stream<...>>
                .map(f -> f.collect(Collectors.toSet()));

        if (result.isPresent()) {
            this.data = result.get();
            this.isInitialized = true;
        } else {
            this.isInitialized = false;
        }
    }
}
```

Figure 7.9: Implementation of the *IpProxyContext*.

7.3 Risk Evaluator

A suitable example for the risk evaluator implementation is the specific *LoginFailuresRiskEvaluator*, which statically evaluates the risk based on login failures shown in Figure 7.10. For further evaluation, the current IP address is required, so it is obtained from the *IpAddressContext*.

The evaluated risk is returned as *Optional*, as the risk might not be evaluated in time or at all. It helps to avoid working with *null* values. This evaluator requires to have information about the authentication user, so the method *requiresUser*() is amended based on it.

```
public class LoginFailuresRiskEvaluator implements RiskEvaluator {
    private final KeycloakSession session;
    private final IpAddressContext ipAddressContext;
    private Double risk;

    public LoginFailuresRiskEvaluator(KeycloakSession session) {
        this.session = session;
        this.ipAddressContext = ContextUtils.getContext(session, DefaultIpAddressFactory.PROVIDER_ID);
    }

    @Override
    public Optional<Double> getRiskValue() {
        return Optional.ofNullable(risk);
    }

    @Override
    public boolean requiresUser() {
        return true;
    }
```

Figure 7.10: Login Failures Risk Evaluator.

The evaluation of the risk itself is done in the *evaluateRisk*() method, which contains a few checks of login failures, as shown in Figure 7.11. As was mentioned before, the evaluation of the risk itself might be challenging and requires specific agreements on the evaluations, as there is no explicit correct approach to achieving it.

In the example, the risk is increasing based on the count of login failures, which might represent a brute-force attack. The higher the count of login failures, the higher the risk.

The last check verifies the current IP address and the IP address of the last login failure, as requesting resources from different IP addresses should indicate a higher risk. There might be a situation

where the risk for a legitimate user trying to access the application will be higher, as some fraudulent activity has been executed based on their account.

```
// Number of failures
var numFailures : int = loginFailures.getNumFailures();
if (numFailures <= 2) {
    this.risk = Risk.NONE;
} else if (numFailures <= 5) {
    this.risk = Risk.SMALL;
} else if (numFailures < 10) {
    this.risk = Risk.MEDIUM;
} else if (numFailures < 15) {
    this.risk = Risk.INTERMEDIATE;
} else {
    this.risk = Risk.HIGH;
}

// Different IP address
if (!currentIp.equals(lastIpFailure)) {
    this.risk = Math.max(risk, Risk.INTERMEDIATE);
}
```

Figure 7.11: Login Failures Risk Evaluator Evaluations.

7.3.1 Configuration

By default, for every risk evaluator, attributes *isEnabled* and *weight* can be configured by the administrator, as described in Section 6.3.1. The permanent storing of the configuration of risk evaluators is done via realm attributes, which are represented by a key-value map structure. Every configurable attribute of the risk evaluator has a unique key that is used to reference it in the map structure.

As shown in Figure 7.12, the attributes are obtained from a different location – from the realm attributes. In this case, if the *weight* attribute is not stored yet or not configured by the administrator, the default value *Weight.IMPORTANT* (weight 0.8) is applied. When the *isEnabled* attribute is not stored or configured, it is turned on by default.

```java
@Override
public double getWeight() {
    return EvaluatorUtils.getStoredEvaluatorWeight(
            session,
            LoginFailuresRiskEvaluatorFactory.class,
            Weight.IMPORTANT
    );
}

@Override
public boolean isEnabled() {
    return EvaluatorUtils.isEvaluatorEnabled(
            session,
            LoginFailuresRiskEvaluatorFactory.class
    );
}
```

Figure 7.12: Login Failures Risk Evaluator Configuration.

7.4 Risk Engine

The implementation of the risk engine relies on leveraging *SmallRye Mutiny* [1] library to comply with the requirements and design introduced in Section 6.3.2. Quarkus officially provides support for the library, and all mentions of the *SmallRye Mutiny* will be primarily focused on the Quarkus Mutiny extension.

The risk engine gathers all risk evaluators, sends a signal to evaluators to start evaluating, retrieves individual scores, and calculates the overall risk score for the authentication process.

In order to improve performance around risk evaluations, an asynchronous approach for the processing has been introduced, as it has minimal impact on the user experience. For more details, see the Section 7.4.1.

The implementation of the overall risk evaluation is described more in detail in Section 7.4.4, the retry and timeout mechanism in Section 7.4.2 and Section 7.4.3.

1. https://smallrye.io/smallrye-mutiny/latest/

7.4.1 Asynchronous Processing

For the risk engine, the more risk evaluators are considered, the more information about the authentication context can be evaluated. However, there might be a high number of evaluators, and their processing of them might be resource and time-consuming. Even the remote evaluators and remote user contexts are considered, so the latency is high, and the current thread is blocked. Asynchronous processing mitigates these problems.

Asynchronous processing of risk evaluators is provided via the *SmallRye Mutiny* library, in which the main concepts are based on asynchronous message passing and non-blocking I/O.

Asynchronous message passing and non-blocking I/O are fundamental elements in building responsive, resilient, and efficient distributed systems. In reactive systems, components communicate asynchronously through message passing, promoting loose coupling in both space and time.

This approach allows systems to handle failures gracefully and adapt to changing conditions. Quarkus utilizes non-blocking I/O to efficiently manage input/output operations, such as database interactions and remote service calls. [34]

The individual risk evaluations are processed on I/O threads, which provides the possibility of using non-blocking I/O. When a remote service is called, the thread is not blocked. During that case, the I/O continuation is scheduled, a continuation is passed, and the thread is released in order to process different tasks. The continuation is basically a code, which will be executed once the I/O is complete.[34]

As shown in Figure 7.13, the evaluator x initiates the risk evaluation, and once it starts executing I/O operations, the I/O continuation is scheduled. During that, as the thread is released, the evaluator y started the risk evaluation. Once the I/O is completed for the evaluator x, the continuation is executed. It utilizes CPU resources as much as possible. In order to guarantee an upper bound time, the timeout is set so evaluations do not spend more time on evaluations than the specified timeout.

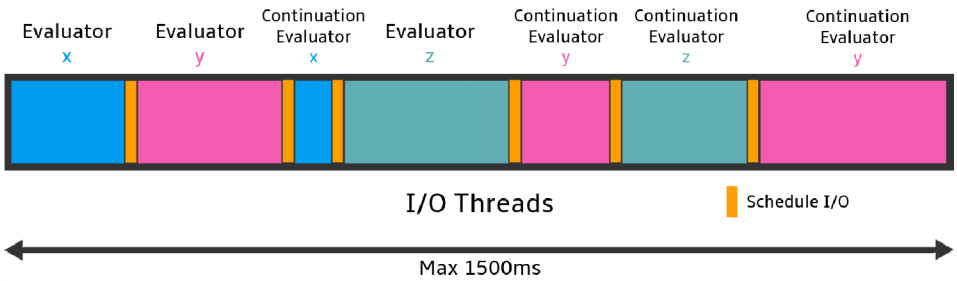

Figure 7.13: *I/O* Threads.

Authentication Flow

As described in Section 6.3.2, two categories of risk evaluators are present for the overall risk evaluations based on the user requirements – ones that require an authentication user and ones that do not. Consider a simple authentication flow with username and password forms and a risk engine authenticator for the additional authentication executions, as shown in Figure 7.14.

In order to not block the main thread with the risk evaluations that do not require a user, a separate thread is used. The risk is not needed at that time at all, so it does not have to slow down the authentication process and get a worse user experience. The dedicated *risk-engine* thread executes tasks for these risk evaluators, leverages the asynchronous processing, and stores the computed overall risk for the particular phase.

The approach for later evaluations of risk evaluators that do require a user is slightly different. The asynchronous approach is used as well, but the risk engine execution needs to be done on the main thread in a blocking manner. The reason behind this is the risk engine needs to have valid, correct results for the overall risk evaluation and, based on the used risk level determination, assess the authentication flow. Therefore, after providing the password, the risk engine authenticator executes risk evaluations, waits for the results, and the authentication flow can proceed.

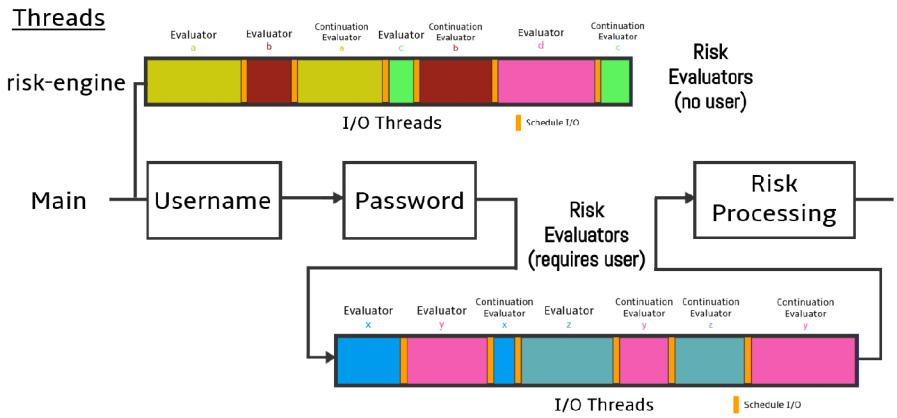

Figure 7.14: Asynchronous Processing Flow.

7.4.2 Retry Evaluations

Due to unexpected failures, individual risk evaluations might not be properly executed. During the risk evaluation, either obtaining user contexts or directly the evaluation might fail. In that case, a retry mechanism has been introduced so the evaluation process is executed again. The management around retrying risk evaluations handles the risk engine, described in Section 6.3.2. The evaluation can be retried at most n times, where n is the specified *retry* parameter described in Section 6.3.2. The visual representation of the case is in Figure 7.15.

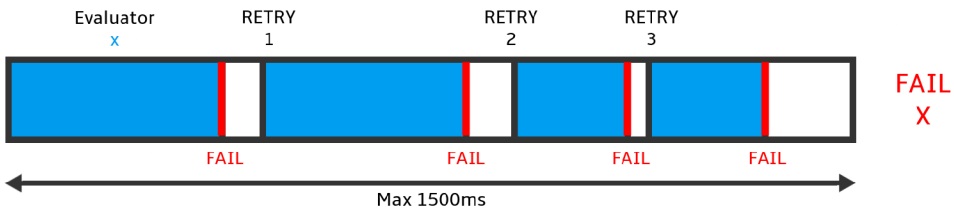

Figure 7.15: Risk Evaluators Retry Mechanism.

7.4.3 Evaluations Timeout

In order to guarantee specific time boundaries, the timeout for the risk evaluators has been introduced. It provides the ability to stop risk evaluation after exceeding a specific configurable timeout for a particular risk evaluator. This means that after exceeding the time limit, the risk evaluator is not taken into consideration for the overall risk score calculations. For more information about the configuration, see Section 6.3.2. The visual representation of the case is in Figure 7.16.

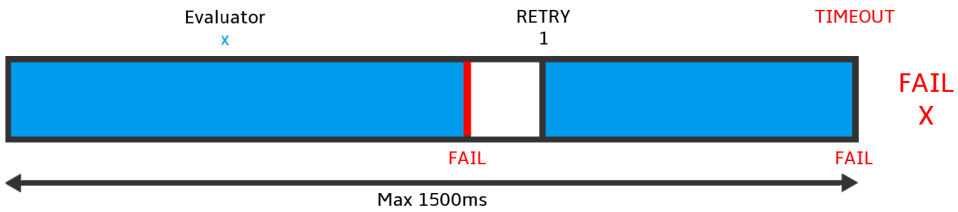

Figure 7.16: Risk Evalutors Timeout.

7.4.4 Evaluation

The overall risk score evaluation made by the risk engine leverages the weighted arithmetic mean algorithm, which is the default one, as described in Section 6.3.2. When all risk evaluators complete the evaluation, results are collected, and the algorithm is applied, as seen in code snippet 7.17.

First of all, for every risk evaluator, the weighted risk score is calculated as $risk * weight$. All of these results contribute to the overall weighted risk score summation, which represents the numerator of the algorithm equation. To obtain a denominator of the algorithm equation, all risk evaluator weights are aggregated and summed together.

After retrieving these two parts of the fraction, the overall risk is calculated and stored for the risk level determination.

```
evaluatedRisks.subscribe().with(risks -> {
    var weightedRisk : double = risks.stream()
            .filter(g -> g.getRiskValue().isPresent())
            .mapToDouble(g -> g.getRiskValue().get() * g.getWeight())
            .sum();

    var weights : double = risks.stream()
            .mapToDouble(RiskEvaluator::getWeight)
            .sum();

    // Weighted arithmetic mean
    this.risk = weightedRisk / weights;

    storedRiskProvider.storeRisk(risk, riskPhase);
});
```

Figure 7.17: Risk Engine Evaluation.

7.5 Artificial Intelligence

The implementation of AI relies on the Natural Language Processing (NLP) engine, as described in Section 6.4. The main purpose of integrating AI into risk-based authentication is for individual risk evaluations.

The default NLP engine is ChatGPT, developed by OpenAI as its popularity rapidly grows and meets all requirements of this thesis. OpenAI provides a REST API for communication with the ChatGPT service, so the integration is seamless.

As described in Section 6.4, the most important properties of the AI NLP engine are:

- *context* – general context for the AI NLP engine to understand the overall meaning and defining role and scope.
- *message* – specific description of the task.

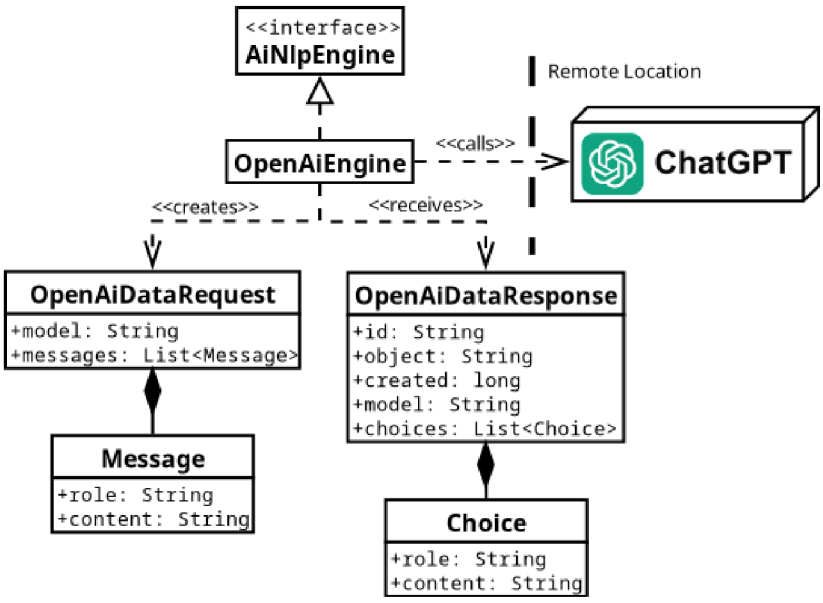

Figure 7.18: OpenAI Integration.

As seen in Figure 7.18, the *OpenAiEngine* implements the *AiNlpEngine* interface and manages the request and response data. ChatGPT API requires specific data to be provided in order to comply with the API. *OpenAiEngine* creates the *OpenAiDataRequest* object, which represents request data sent to the API.

The *context* and *message* properties are passed to the request data as *Message* objects. The *context* property is represented as *Message* object with role *system* and content *context*. The *message* property is represented as *Message* object with a role *user*, and content *message*. The data for evaluating risk score are represented as a JSON object with properties *risk* and *reason*, as explained in Figure 7.19.

The default context for evaluating risk score with all necessary information about tasks, boundaries, and format of the result might be as in Figure 7.19.

```
Evaluate a risk that the user trying to authenticate is fraud.
Return the double value in the range (0,1>, as f.e. 0.8,
which means an 80% chance of the authentication attempt being very critical.

    - Values close to 0 mean the risk of user fraud is low.
    - Values close to 1 mean the risk of user fraud is high.

Analyze the provided data and return risk values.
The message MUST be in JSON format, with two items - 'risk' and 'reason'.
The 'risk' item MUST contain the evaluated risk double value described above.
The 'reason' item MUST briefly describe the reason why it was evaluated like that.

f.e.
 {
   "risk": 0.7,
   "reason": "Many login failures, with a high probability of brute-force attack."
 }
```

Figure 7.19: AI NLP Context Message.

Specific tasks for AI-based risk evaluators contain a dynamically created message with all necessary information for a device representation, as seen in Figure 7.20.

```
Give me the overall risk that the device trying to authenticate is fraud based on its parameters.
-----
IP Address: 127.0.0.1
Browser: Chrome/124.0.0
Operating System: Linux
OS version: Unknown
Is Mobile? : false
Last access: 0
-----
```

Figure 7.20: AI NLP Device Query.

The response from the ChatGPT contains the required data with the specific risk score and the explanation of the reason why it was evaluated like that, as seen in Figure 7.21.

```
{
  "risk": 0.2,
  "reason": "Low risk of fraud as the IP address is local, common browser and operating system,
   not mobile, and no recent activity."
}
```

Figure 7.21: ChatGPT Device Response.

The other example is related to evaluating login failures with the user *message*, as seen in Figure 7.22.

```
Give me the overall risk that the user trying to authenticate is a fraud based on its parameters.
These parameters show the metrics about login failures for the particular user.
Used for detection of brute force attacks.
After each successful login, these metrics are reset.
-----
Number of login failures for the user: 12
IP address of the last login failure: 222.0.0.15
Current device IP address: 222.0.0.10
Number of temporary lockouts for the user: 3
Last failure was before: 1234 ms
-----
```

Figure 7.22: AI NLP Login Failures Query.

7.6 External Integration

To demonstrate the external integration with third-party services, the external location user context has been introduced. As described in Section 6.5, it is possible to obtain remote user context of arbitrary type. Obtaining more comprehensive information about the location of the authentication device might be challenging and requires the incorporation of services concentrated on it. Therefore, retrieving location information from the remote service is suitable for demonstrating external integration with services.

For this purpose, the *ipapi*[2] service is used as it provides a simple API for location information based on the provided IP address. Remote calls on the service are executed during the initialization of the *IpApiLocationContext* when the IP address from the *IpAddressContext* is returned and substituted to the request.

The approach for the request assembly is straightforward and uses only path parameters of the URL as follows: *https://ipapi.co/<ip-address>/json*. The data in the format specified in *IpApiLocationData* class is returned. The simplified architecture of the *IpApiLocationContext* context is shown in Figure 7.23.

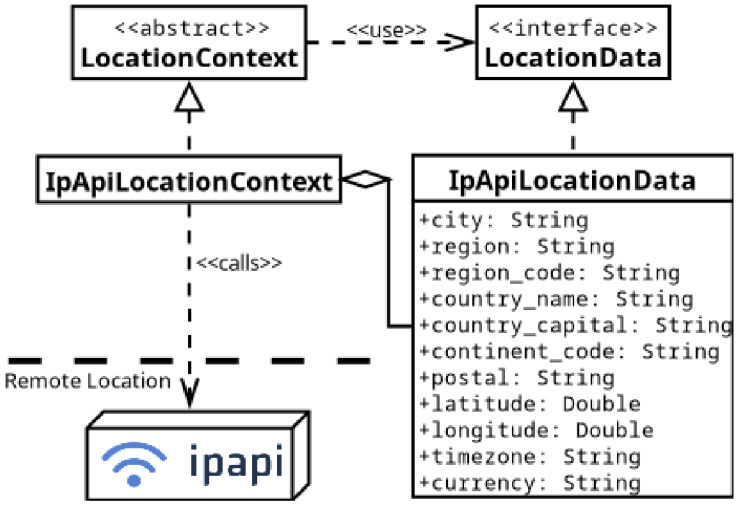

Figure 7.23: External Service *ipapi*.

2. https://ipapi.com/

8 Conclusion

This thesis has comprehensively explored the concept of adaptive authentication, focusing on its implementation in Keycloak. The need for adaptive authentication arises from the inadequacies of traditional methods in addressing the dynamic nature of cybersecurity threats. By adjusting authentication requirements based on user contexts and behavior, adaptive authentication offers a robust solution for enhancing security.

A clear understanding of aspects and requirements has been gained after comprehensive research in the adaptive authentication area and a comparative analysis of existing solutions. Keycloak has proven to be a very suitable platform for implementing adaptive authentication due to its flexible architecture and comprehensive features. The extendability of the Keycloak capabilities was done almost entirely separately from the server, besides extending the administrator console with non-standard user interface components for the management of authentication policies. Other than that, I was surprised there were no limitations concerning the extension of Keycloak capabilities.

The architecture for achieving the required aspects has been reconsidered a few times due to the simplification of the approach or as specific issues related to the design occurred. To be more specific, during the implementation, I recognized providing accidental complexity[35], as the design of components for risk-based authentication introduced several additional abilities that were not essential for solving the problem. It consisted of added attributes, operations, or relations between components. They did not bring any extra value, so they have been simplified for better manageability.

The implementation chapter showed how to smoothly integrate adaptive authentication mechanisms into Keycloak, highlighting the importance of the role of a policy engine, risk evaluators, and the use of AI to improve security decisions.

In conclusion, I believe this thesis might significantly contribute to the field of adaptive authentication by offering many relevant conceptual insights and practical implementation strategies using Keycloak. During the work on the thesis, I understood there might not be an optimal solution for a problem, but many with certain deficiencies

need to be considered. I have learned a lot about the Keycloak internal codebase and important authentication aspects. Such a comprehensive topic required a lot of planning and brainstorming, as during that, I used common techniques for software development and deepened my practical usage of them.

Future research could explore further advancements in AI integration and the development of more sophisticated adaptive algorithms to counter emerging cybersecurity threats. I feel the Keycloak community might be interested in this approach as it has a certain potential. I want to continue developing it and bring closer a brighter future for adaptive authentication and Keycloak.

Bibliography

1. *Keycloak - Open Source Identity and Access Management* [online]. 2024. [visited on 2024-05-17]. Available from: https://www.keycloak.org/.
2. *Adaptive Authentication* [online]. [visited on 2024-05-15]. Available from: https://www.logintc.com/types-of-authentication/adaptive-authentication/.
3. *What is Adaptive Authentication ?: Adaptive Authentication vs. Traditional Authentication* [online]. [visited on 2024-05-15]. Available from: https://www.silverfort.com/glossary/adaptive-authentication/#adaptive-authentication-vs-traditional-authentication-pros-and-cons.
4. *What Is Adaptive Authentication (and When to Use It)* [online]. [visited on 2024-05-15]. Available from: https://www.descope.com/learn/post/adaptive-authentication#benefits-and-drawbacks-of-adaptive-authentication.
5. *What is Adaptive Authentication and its advantages?* [online]. [visited on 2024-05-15]. Available from: https://www.incognia.com/the-authentication-reference/what-is-adaptive-authentication.
6. *5 Authentication Factors: A Guide From Passwords to Biometrics* [online]. [visited on 2024-05-15]. Available from: https://www.aratek.co/news/5-authentication-factors-a-guide-from-passwords-to-biometrics.
7. *What Is Multi-Factor Authentication (MFA)?* [online]. [visited on 2024-05-15]. Available from: https://www.descope.com/learn/post/mfa.
8. *What Are the Three Authentication Factors?* [online]. [visited on 2024-05-15]. Available from: https://rublon.com/blog/what-are-the-three-authentication-factors/.
9. *Authentication factor - definition overview: Five authentication factor categories and how they work* [online]. [visited on 2024-05-15]. Available from: https://www.sumologic.com/glossary/authentication-factor/.

10. STEWART, James Michael. The three types of multi-factor authentication(MFA): What is multi-factor authentication? [online]. 2018 [visited on 2024-05-15]. Available from: https://www.globalknowledge.com/us-en/resources/resource-library/articles/the-three-types-of-multi-factor-authentication-mfa/.

11. *Okta Adaptive Multi-Factor Authentication* [online]. [visited on 2024-05-15]. Available from: https://www.okta.com/sites/default/files/okta_mfa-datasheet.pdf.

12. *Customize Adaptive MFA: Examples of low-risk, high-confidence scenarios* [online]. [visited on 2024-05-15]. Available from: https://auth0.com/docs/secure/multi-factor-authentication/adaptive-mfa/customize-adaptive-mfa#examples-of-low-risk-high-confidence-scenarios.

13. *Customize Adaptive MFA: Confidence scores* [online]. [visited on 2024-05-15]. Available from: https://auth0.com/docs/secure/multi-factor-authentication/adaptive-mfa/customize-adaptive-mfa#confidence-scores.

14. *Org-level security: Risk scoring* [online]. [visited on 2024-05-15]. Available from: https://help.okta.com/oie/en-us/content/topics/security/security_risk_scoring.htm.

15. *Org-level security - ThreatInsight: System Log events for Okta ThreatInsight* [online]. [visited on 2024-05-15]. Available from: https://help.okta.com/en-us/content/topics/security/threat-insight/configure-threatinsight-system-log.htm.

16. WIKIPEDIA CONTRIBUTORS. *Citrix Systems — Wikipedia, The Free Encyclopedia* [online]. 2024. [visited on 2024-05-17]. Available from: https://en.wikipedia.org/w/index.php?title=Citrix_Systems&oldid=1201987825.

17. RAGHAVAN, Vijaya. Introducing Citrix Adaptive Authentication [online]. 2022 [visited on 2024-05-15]. Available from: https://www.citrix.com/blogs/2022/06/01/introducing-citrix-adaptive-authentication/.

18. *Device Posture* [online]. 2024-02-22. [visited on 2024-05-15]. Available from: https://docs.citrix.com/en-us/device-posture.html.

19. *What is Adaptive Authentication?* [online]. [visited on 2024-05-15]. Available from: https://www.onelogin.com/learn/what-why-adaptive-authentication.

20. *Vigilance AI™ Threat Engine* [online]. [visited on 2024-05-15]. Available from: https://www.onelogin.com/product/vigilance-ai.

21. *Get Score Insights* [online]. [visited on 2024-05-15]. Available from: https://developers.onelogin.com/api-docs/2/vigilance/get-scores.

22. *Get a Risk Score* [online]. [visited on 2024-05-15]. Available from: https://developers.onelogin.com/api-docs/2/vigilance/verify.

23. *Risk-Based Authentication* [online]. [visited on 2024-05-17]. Available from: https://www.silverfort.com/use-cases/risk-based-authentication/.

24. *Adaptive Authentication* [online]. [visited on 2024-05-15]. Available from: https://www.silverfort.com/glossary/adaptive-authentication.

25. *Adaptive Authentication: How can adaptive authentication detect that a user account is compromised?* [online]. [visited on 2024-05-15]. Available from: https://www.silverfort.com/glossary/adaptive-authentication/#adaptive-authentication-vs-traditional-authentication-pros-and-cons.

26. *What Is Risk-Based Authentication?* [online]. 2021-01-09. [visited on 2024-05-15]. Available from: https://www.beyondidentity.com/blog/what-risk-based-authentication.

27. *Introduction to the Service Provider Interfaces* [online]. Oracle, 2024. [visited on 2024-05-17]. Available from: https://docs.oracle.com/javase/tutorial/sound/SPI-intro.html.

28. *Keycloak - Configuring providers* [online]. 2023. [visited on 2024-05-17]. Available from: https://www.keycloak.org/server/configuration-provider.

29. *Keycloak - Server administration - Authentication flows* [online]. 2023. [visited on 2024-05-17]. Available from: https://www.keycloak.org/docs/latest/server_admin/#_authentication-flows.

30. GILLIS, Alexander S. What is a proof of concept (POC)? [online]. 2023 [visited on 2024-05-15]. Available from: https://www.techtarget.com/searchcio/definition/proof-of-concept-POC.

31. *Core J2EE Patterns - Data Access Object* [online]. Oracle, 2001-2002. [visited on 2024-05-17]. Available from: https://www.oracle.com/java/technologies/dataaccessobject.html.

32. *What is a REST API* [online]. 2020-05-08. [visited on 2024-05-15]. Available from: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

33. MARTIN, Robert C. The Single Responsibility Principle [online]. 2014 [visited on 2024-05-15]. Available from: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html.

34. *Quarkus Reactive Architecture* [online]. [visited on 2024-05-17]. Available from: https://quarkus.io/guides/quarkus-reactive-architecture.

35. BENČEVIĆ, Marin. Accidental and Essential Complexity — Programming Word of the Day [online]. 2018 [visited on 2024-05-17]. Available from: https://medium.com/background-thread/accidental-and-essential-complexity-programming-word-of-the-day-b4db4d2600d4.