UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Authentication and Authorization in Blind Data Miners

Morten Myrland

Master Thesis in Computer Science - INF-3981

69° 40’ 53.7" N
18° 58’ 37.1" E

“Bad decisions make good stories.”
–Ellis Vidler

“If no one comes from the future to stop you from doing it, then how bad of a decision can it really be?”
–Unkown

Abstract

Chronic pain is defined as pain that lasts for at least 12 weeks. People with chronic pain conditions can have difficulties getting through daily tasks because the pain can limit their mobility, strength, and endurance.

As of 2021, there is no universal treatment that works for all cases of chronic pain. A tool that can give personalized treatment alternatives for each patient can benefit this group of patients significantly.

This thesis is a part of the chronic pain research project at the Norwegian Centre for E-health Research, where they make a privacy-preserving distributed storage system called blind data miners. This system will store and compute statistics on patient-reported outcomes and experiences from treatments on chronic pain. The data is collected directly from patients via a mobile app and Fitbit. The data can then be used by health workers to give personalized treatments to chronic pain patients.

Several reports and studies have shown that almost every health app on the market is vulnerable to API attacks in some way. Health apps store highly sensitive data, so this data must be protected from unauthorized access.

This thesis is looking at a decentralized alternative for authentication and authorization in blind data miners. This alternative is implemented and evaluated according to a set of requirements. Based on this, the thesis concludes with a discussion on whether the proposed alternative is viable for use in blind data miners.

Acknowledgements

Dear everyone,

I would like to thank my supervisors Professor Randi Karlsen and Professor Johan Gustav Bellika. Your feedback on this thesis has been invaluable.

To my family. Thank you for giving me unconditional support on whatever I do.

To my friends at office A122. Isak, Joakim, Magnus and Eric. Thank you for all discussions we have had. Both meaningful and less meaningful. This would have been impossible without you.

To all my other friends at the university. Thank you for keeping me sane throughout this process. No names mentioned, but you know who you are. Thank you for being you.

With love,
Morten

Contents

List of Figures

List of Tables

1 Introduction

A recent document from the Norwegian Directorate of Health looks at the potential that mHealth apps has. The data collected by mHealth apps has clinical value and are useful for healthcare professionals and for research. The COVID-19 pandemic has shown the importance of having safe and effective digital tools to be able to check up on patients remotely[12][23].

The global mHealth apps market size was valued at around 40 billion U.S dollars in 2020 and is expected to grow at a compound annual growth rate (CAGR) of 17.7% from 2021 to 2028 according to a report by Grand View Research[23]. This means the market size in 2028 is estimated to be 149 billion U.S dollars[23]. Figure 1.1 shows the U.S mHealth apps market size from 2016 - 2028.

There are over 327,000 mHealth apps on the global market. Patients and healthcare professionals mostly rely on reviews made by other users when deciding whether they should use one of these apps[12]. ORCHA, an independent company located in London, does extensive reviews of mHealth apps, and according to them, 85% of mHealth apps found in app stores does not meet their minimum quality requirements which includes security requirements[12].

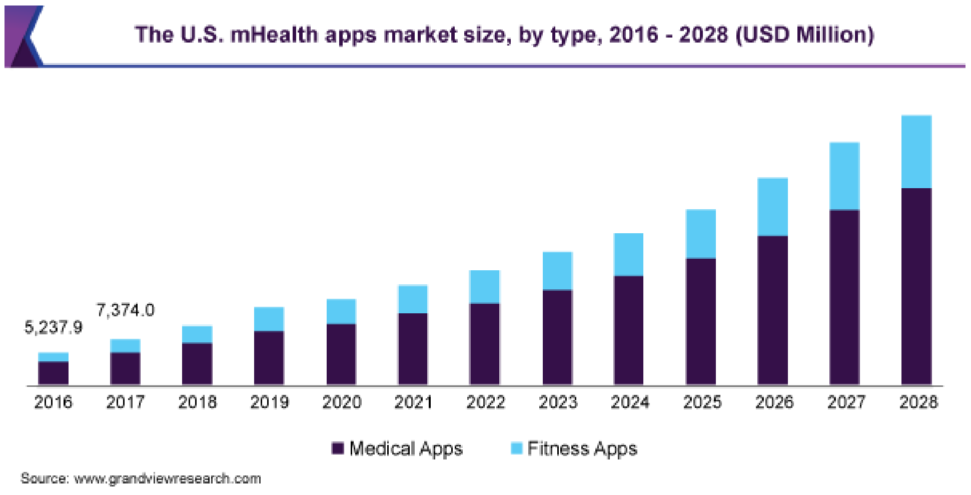


Figure 1.1: U.S mHealth Market Size 2016 - 2028[23]

Data stored by health apps can be very sensitive, and it is therefore crucial that security is highly prioritized by the developers of these apps.

From 2009 to 2013 there were close to 27 million individuals in the U.S affected by data breaches involving personal health data[29]. A recent study published by cybersecurity firm Approov tested 30 of the leading mHealth apps and found that all of them were vulnerable to API attacks which could allow unauthorized individuals to gain access to full patient records[17]. We need to address the lack of security in mHealth and e-Health in general if we are going continue going forward with the adoption of these technologies.

1.1 Project Requirements

The Chronic Pain project is an ongoing project at the Norwegian Center for E-Health Research (NSE) where the goal is to build a system that can compute statistics on patient reported outcomes and experiences from pain treatments without learning any personal details about individual patients. Healthcare workers can then give personalized treatments to patients based on the symptoms that the patient is exhibiting and the statistics from similar cases.

The patients' data are stored in Blind Data Miners (BDM). BDMs are servers that store data that has been split using Shamir's secret sharing, a technique that split data in such a way that the data can only be read when you have all shares from a split. BDMs will be hosted by independent organizations.

The data stored in these blind data miners are very sensitive and must not be open to the public. Patients are the owner of this data, and they should be the ones deciding who has access to their data.

Access control in computer systems are managed by an identity and access management (IAM) system.

One of the ideas of using BDMs is that it distributes the data so that attackers must breach all servers that are hosted by different organizations to get the shares that are needed to reconstruct the original data. To avoid having a centralized point of attack, the IAM system should ideally be distributed. A centralized IAM server that gets attacked or accidentally leak usernames and passwords could compromise all patients that store their data in BDMs.

In centralized IAM servers, the server administrators can see every permission that every user has. They can also edit, delete or add permissions as they please. To keep the privacy-preserving properties of BDMs throughout the whole system, the IAM system should only reveal permission policies to those who need them, and only those who grant a permission should be allowed to edit or remove it.

Patients can not be expected to be online at all times. The system must allow offline participants to discover permissions that were delegated to them when they were offline. This is trivial to solve in a centralized system, but not in a decentralized one.

It is also important that the IAM system is highly available. Without it, data will not be accessible. It must also be scalable to handle an increasing number of users. Finally, the system must be fast and responsive or else it will be frustrating to use.

Below is a list that summarizes the requirements.

- Decentralized.
- Authorization delegation.
- Fast, scalable and highly available
- Private permission policies.
- Offline participants.

The chronic pain project is currently in development with a centralized Keycloak server for IAM.

1.2 Goal

The goal of this thesis is to look for a decentralized alternative for identity and access management that will be used with blind data miners in the chronic pain project.

When a suitable alternative has been found, a prototype will be designed, implemented and evaluated according to the requirements specified in section 1.1

1.3 Contribution

This thesis has identified requirements for a decentralized IAM system for use in blind data miners in the chronic pain project. Based on these requirements, relevant literature has been investigated to find a suitable alternative to the centralized solution with Keycloak that is currently in development.

The alternative solution is demonstrated via an implementation. Finally, the solution is evaluated to see if it is a feasible alternative to Keycloak. This will help the researchers make an informed decision when choosing an IAM system.

1.4 Method

It is essential to have a structured plan when conducting academic research to achieve proper, correct, and well-founded results. The work of [11] presents methods and methodologies for research projects and degree projects, and help define a path for what methods to use.

This thesis present a *qualitative* research by investigating an alternative to the currently chosen IAM system used with BDMs. This alternative will be implemented and evaluated to see if it is a better option than the one in development today.

The research method adopted will be an *applied* method, which involves solving a known and practical problem. The design and implementation of an IAM system is such a problem. The thesis builds on existing research on cryptography and privacy[26][31][24].

1.5 Thesis Outline

- **Chapter 2** covers necessary background theory for this thesis.
- **Chapter 3** describes the research methods used.
- **Chapter 4** describes a centralized IAM design that is currently in development for the chronic pain project. It also proposes an alternative decentralized design.
- **Chapter 5** describes the implementation of the decentralized design proposed in chapter 4
- **Chapter 6** evaluates the implemented decentralized design.
- **Chapter 7** discusses advantages, limitations, and future work of the implemented design. It also discusses the design's viability compared the the centralized solution that is in development by NSE.
- **Chapter 8** concludes the thesis.

2 Background

This chapter covers concepts and technologies necessary to understand the thesis.

Chapter Outline

- **Section 2.1** gives a short description on the chronic pain project. This is the project that the design and implementation is intended for.
- **Section 2.2** describes blind data miners. Blind data miners are at the core of the chronic pain project.
- **Section 2.3** gives a short explanation on what an identity and access management system is, and why it is needed.
- **Section 2.4** describes Merkle hash trees. This is an important prerequisite for understanding how certificate transparency works.
- **Section 2.5.1** describes certificate transparency. This is described because WAVE, an authorization framework used in this thesis, has developed a storage solution that is an extension of certificate transparency.
- **Section 2.6** describes WAVE, which is an authorization framework that the proposed alternative solution for IAM is built around.

2.1 Chronic Pain Project

The Norwegian Center for e-Health Research (NSE), is currently working on a project where the goal is to provide personalized treatment decisions to patients suffering from chronic pain. The project will provide patients with a Fitbit for automatic data collection, and a mobile app where patients can manually report the outcome and experiences from different treatments[5]. This data is distributed across a number of blind data miner(BDM, see section 2.2) servers. The BDMs enable an analyst to execute statistical queries on the data without learning anything about individual patients[31]. The idea is that these statistics will be used to create personalized treatments based on patients' symptoms

2.2 Blind Data Miners

This section will be a short description of blind data miners. The technical bits and pieces that make them work are beyond the scope of this thesis. This is just a general overview.

Blind data miners uses an algorithm developed by Adi Shamir in 1979 which has later become known as Shamir's Secret Sharing. It is a technique for splitting data into *shares* that, by themselves, does not reveal anything about the original data. The data is split by utilizing some properties of polynomials that allow the original data to be reconstructed with the help of Lagrange Interpolation. To reconstruct the data, a specified threshold number of shares are needed. Details on the algorithm can be found in the original paper by Adi Shamir[26], but it is not necessary to know for this thesis.

A blind data miner is a server that stores shares of data that has been split using Shamir's Secret Sharing. Each server only holds one share of the data and is therefore blind to the original data, hence the name blind data miner[31].

2.3 Identity and Access Management

Identity and Access Management (IAM) is about defining and managing roles and privileges of users in a computer system. Users can be customers, employees, or even other computer systems. The core objective of IAM systems is to provide a digital identity per individual, and give those individuals access to the right resources in the right context.

Typically, administrators of IAM systems have tools to change a user's role, track user activities, and enforce policies on an ongoing basis.[19]

Authentication

The identity management in an IAM system is an authentication concern. Authentication is the determination of the identity or role that someone has. This determination can be done in a number of different ways, but it is usually based on a combination of three factors. These authentication factors are something you are (e.g finger print and/or iris), something you have (e.g keycard), and something you know (e.g a password) [21][27].

Authorization

Authorization is about access management. Authorization determines if a person or system is allowed access to resources, based on an access control policy. Such authorizations should prevent an attacker from bypassing the system entirely, or tricking it into giving them access [21].

2.4 Merkle Hash Tree

A Merkle tree or hash tree is a tree where leaf nodes are the hash of some data. Non-leaves are the hash of its children nodes concatenated. Merkle trees are used to verify the contents of large amounts of data efficiently. Figure 2.1 shows a Merkle tree for data blocks L1, L2, L3 and L4. If this is data in a distributed system with replica databases, it is important that these databases contain the same data. If all replicas have their own Merkle tree, a simple comparison between the root nodes of the trees would be enough to detect inconsistencies. A single corrupt bit in one data block would result in a different hash value for that block which would propagate to the root node.[20].

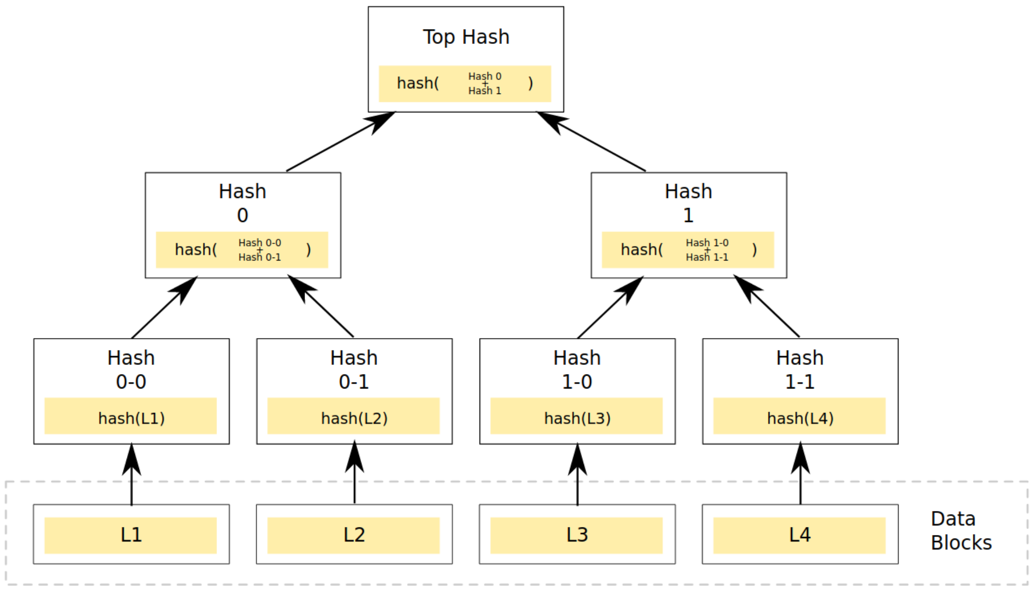


Figure 2.1: Merkle Tree [30]

If root nodes does not match, we can start traversing the tree to find, in O(log n) time, exactly which block has been corrupted.

2.5 Certificate Transparency

Certificate Transparency(CT) is an open framework designed to prevent Certificate Authorities(CA) from maliciously or mistakenly issuing digital certificates. DigiNotar, a former Dutch CA, was hacked in 2011 and issued over 500 fake certificates for multiple domains[33][18]. With CT, fake certificates can be detected almost immediately[15].

2.5.1 Certificate Transparency Overview

Figure 2.2 shows how it works. (1) A domain owner requests a certificate from a CA. (2a) The CA sends a precertificate to the log provider. Precertificates are like regular certificates except that they contain an extension so that user agents (typically browsers) will not accept it. Precertificates help break a deadlock in CT. Before a CA can log a certificate, the certificate needs an SCT (Signed Certificate Timestamp). But for the certificate to get an SCT, it needs to have been submitted to a log. (2b) The certificate is created and appended to a Merkle hash tree, which is the log that enables transparency. (2c) The log responds with a SCT which is a promise that the certificate will be included within a certain timespan (usually 24 hours). (3) The certificate is sent to the domain owner. (4,5) The certificate is presented to visitors as it is in regular non-transparent PKI[14].

(6) Monitors continuously check the logs for suspicious activities. If a log provider is caught trying to manipulate the logs they are permanently removed as an approved log provider[22].

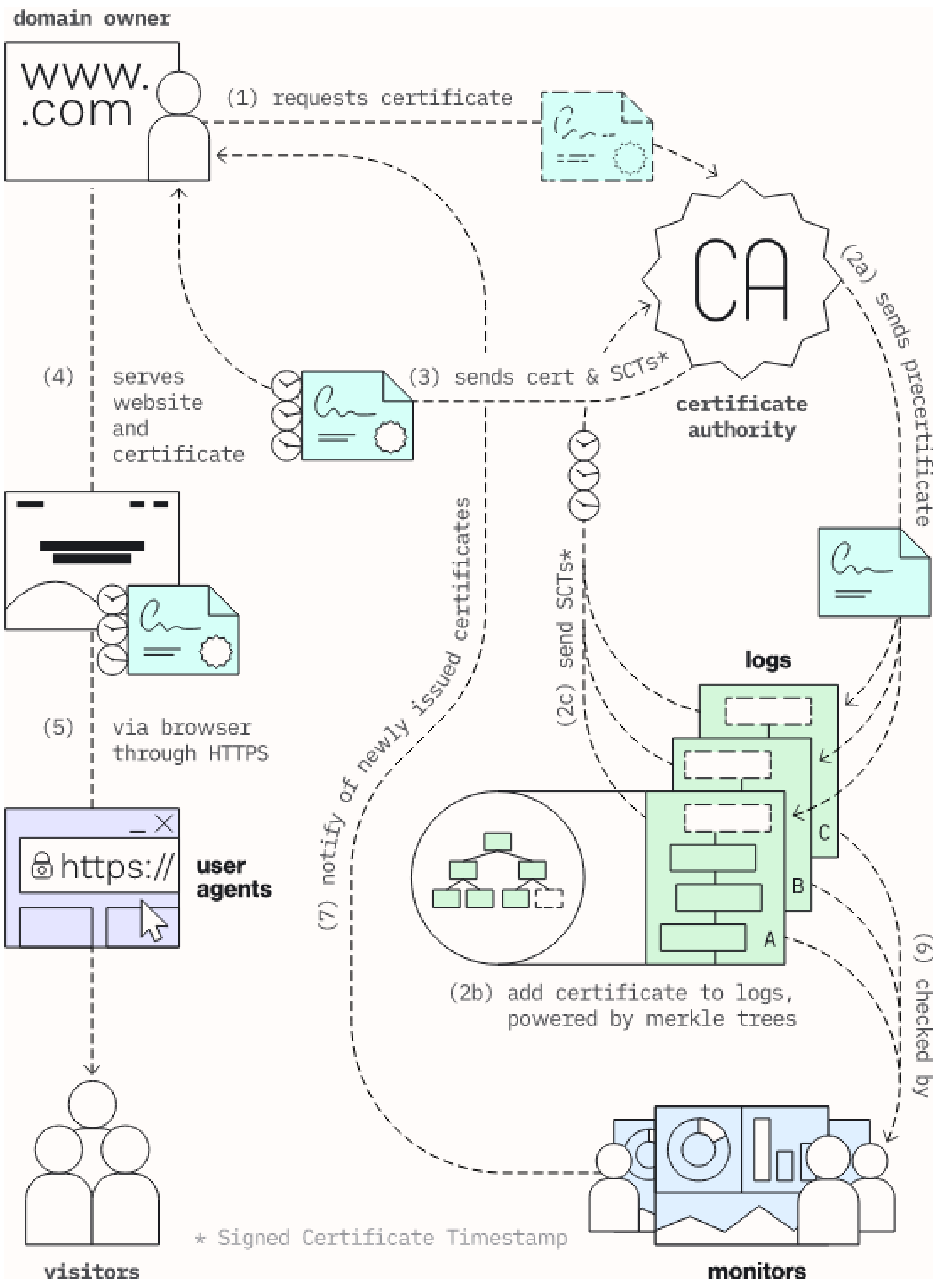

domain owner
www.
.com
(1) requests certificate
CA
certificate
authority
(2a) sends precertificate
(3) sends cert & SCTs*
(4) serves website and certificate
(2c) send SCTs*
(7) notify of newly issued certificates
logs
A
B
C
(2b) add certificate to logs, powered by merkle trees
(6) checked by
(5) via browser through HTTPS
https://
user agents
visitors
monitors
* Signed Certificate Timestamp

Figure 2.2: Certificate Transparency Ecosystem[14]

2.5.2 Monitor

Monitors are publicly run servers that watches the certificate logs for suspicious certificates. They watch for certificates that have unusual extensions or permissions, such as certificates that have CA capabilities. They also check and prove that all certificates has been consistently appended to the log. It can be proved with a Merkle consistency proof.[18]

2.5.3 Auditor

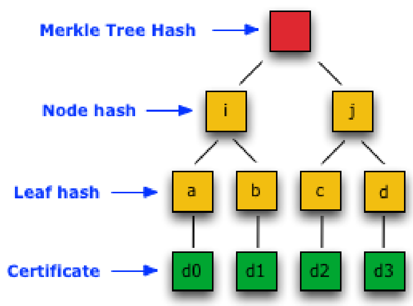
Auditors makes sure that logs are consistent and append only. Today's log should contain everything from yesterday's log. They can also check that a particular certificate is present in the log. Monitors can also be auditors.[18]

2.5.4 Merkle Consistency Proof

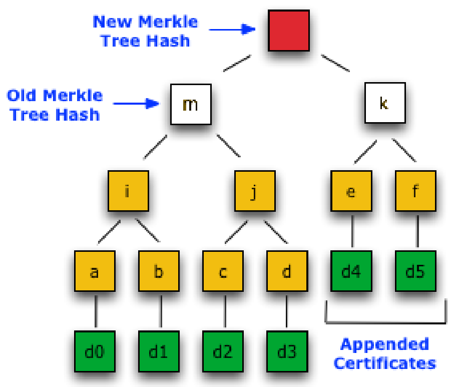
A Merkle consistency proof lets you verify that a later version of a log contains all entries from a previous version plus the new entries. No certificates have been back-dated and inserted into the log, no certificates have been modified, and no certificates have been deleted.

Let us say we want to append two new certificates, $d4$ and $d5$ to the log in figure 2.3a. The consistency proof is worked out as followed. First, we need to verify that the old Merkle tree hash is a subset of the new one. Then we need to verify that the new Merkle tree hash is the concatenation of the old one plus all the intermediate hashes of the newly added certificates. The consistency proof is the minimum number of intermediate node hashes needed to compute these two tings.

In this case we only need m and k. m proves that the old log is a subset of the new one, and with m and k we can prove that the new log is a concatenation of the old log plus the newly added certificates[6][10].



(a) Original log



(b) New log with appended certificates

Figure 2.3: Merkle Consistency Proof A[10]

If you append two more to this (figure 2.3b), the Merkle consistency proof would in this case be three nodes (*m*, *k* and *l*) as shown in figure 2.4

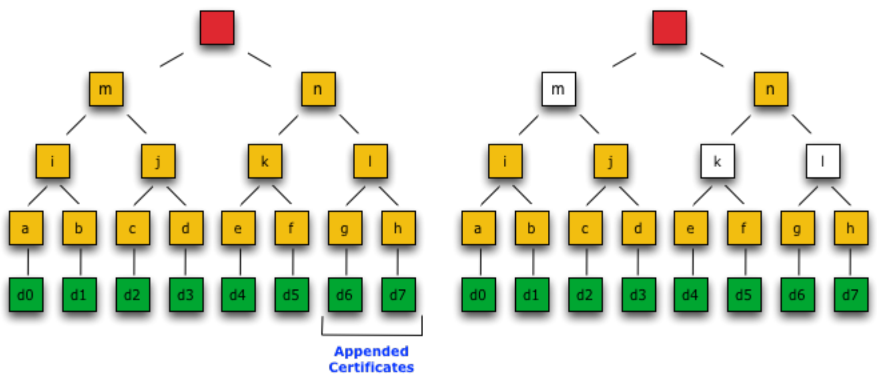


Figure 2.4: Merkle Consistency Proof B[10]

2.5.5 Merkle Audit Proof

An audit proof lets us verify that a specific certificate has been appended to the log. Audit proofs are important because clients should reject any certificates that do not appear in a log.

An audit proof is the minimum node hashes required to compute all of the nodes between the leaf and the tree root. If the tree root we compute matches the log's advertised tree root, then the certificate is in the log.

Let us say that we want to verify that certificate *d3* exists in the log (figure 2.5). We already know *d* because that is the hash of the certificate we want to check. To compute the root hash, we need leaves *i*, *c* and *n*.

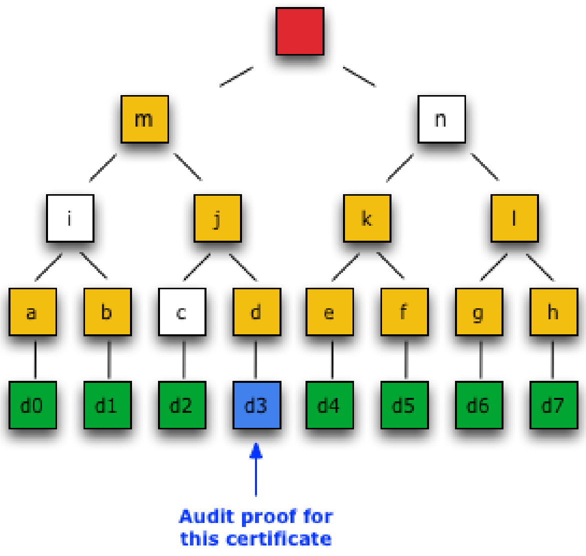


Figure 2.5: Merkle Audit Proof[10]

2.6 WAVE: A Decentralized Authorization Framework with Transitive Delegation

WAVE is an authorization framework where, unlike Keycloak, there is no central authority responsible for issuing access to resources. It is fully distributed, and any participant can delegate portions of their permissions autonomously.

WAVE uses a graph-based authorization model such as in SDSI/SPKI[25][32] where proof of authorization is a path through the graph[4]. See section 2.6.3 for more.

WAVE was chosen over other solutions like SDSI/SPKI and Macaroons[7] because they do fulfill the requirement of offline participants.

2.6.1 Usage scenarios

WAVE has been deployed in over twenty small to medium-sized commercial and residential buildings. The main focus has been securing distributed IoT devices and services to monitor and control these buildings. WAVE is not limited to IoT devices. It provides general-purpose delegable authorization and can, for example, be used in place of OAuth to remove the risk of the centralized token-issuing server and allow for richer delegation semantics. Smart buildings are used as a running example and provide an intuitive understanding of the framework[4].

2.6.2 Terminology

A short explanation of the terminology used in WAVE.

Entity

An entity is a collection of private and public keypairs and can correspond to a user, service, or group.

Policy

A policy is one or more permissions along with a description of the resources for which the permissions are granted and the expiry of the grant.

Attestation

An attestation is a signed certificate containing a policy. Each edge in the graph is an attestation.

Namespace authority

The namespace authority is an entity who is the root of authorization for a resource. It is the entity who has permissions without having received it from someone else.

Proof

A path from an entity to another entity through the graph grants access to the intersection of the policies on that path. Entities can prove they have some permission *P* by revealing a path through the graph from a namespace authority to themselves where all edges have the same permission *P*. This path is a proof.

2.6.3 WAVE Overview

Global Authorization Graph

The global authorization graph in WAVE consists of entities and attestations. Nodes in the graph are entities, and edges are attestations that represent the permission grants between two entities. The client (representing a user, device, or service) interacts through the WAVE service with the global authorization graph. Clients can use the WAVE service to grant permissions to other entities. The WAVE service constructs an attestation signed by the granting entity containing a policy. An attestation consists of:

- **Issuer:** The entity granting the permission.
- **Subject:** The entity receiving permissions.
- **Subject:** A description of the permissions (e.g read), and a URI to the resource the attestation gives access to.
- A signature from the issuer.

When accessing a service, clients request a proof from the WAVE service. This proof is verified by the entity providing the service that is requested. The proofs are generated using a protocol called reverse-discoverable encryption (RDE) [4].

Reverse-Discoverable Encryption

Attestations are encrypted to make sure that they are private. The encryption layer is transparent to the clients. The WAVE service discovers and decrypts the relevant portions of the graph required to form a proof automatically.

Each entity has an additional public/private keypair used for encrypting and decrypting attestations. This keypair is separate from the one used to sign attestations. When an entity grants a permission, it attaches its private key to the attestation and encrypts the attestation (including the attached private key) using the receiving entity's public key.

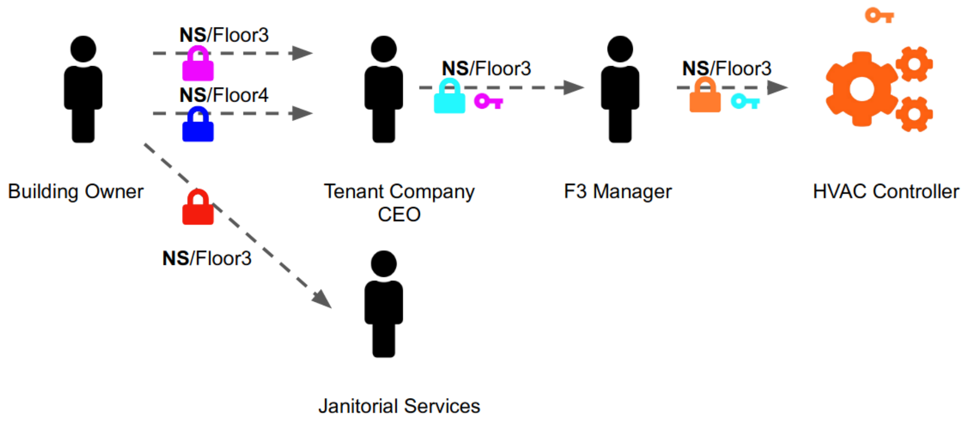


Figure 2.6: Reverse-Discoverable Encryption[3]

Figure 2.6 illustrates how this is done. Recall that WAVE is not limited to IoT devices. In this example, there is a heating, ventilation, and air conditioning (HVAC) controller, which has been granted permission to floor three by the floor three manager (F3 Manager). The F3 manager has been given access to floor three by the company CEO, which has received access from the building owner. When the HVAC controller wants to prove that it has access to floor 3, it has to find a path in the graph from the building owner to itself.

Attestations do not have to be created in order, but for simplicity, the figure's description will be as if they were granted in order. The *building owner* creates an attestation that contains a policy giving access to floor 3 to the *CEO*. The attestation is encrypted using the *CEO's* public key (illustrated by the pink padlock). The *CEO* then gives access to floor 3 to the *F3 Manager*. This attestation is encrypted using the *F3 managers* public key (light blue padlock), and the *CEO's* private key is also added to the attestation (pink key). Finally the *F3 manager* gives access to floor 3 to the *HVAC controller* where the attestation contains *F3 manager's* private key and is encrypted with the *HVAC controller's* public key.

When the *HVAC controller* wants to prove that it has access to floor 3, it uses its private key to decrypt the attestation from *F3 manager*, and at the same time discovers the key to decrypt the next upstream attestation given to the *F3 manager* by the *CEO*. This continues until it reaches the building owner, which is where the permissions originate from. Also known as the namespace authority.

Simply finding a path that through the graph is not secure enough. The RDE has to be policy-aware, meaning proving entities should only be able to decrypt attestations that contain intersecting policies. For example, figure 2.6 shows that the *building owner* has granted access to floor 4 to the *CEO*. This attestation is on the path to the *HVAC controller*, but the permissions on this path do not intersect. WAVE uses a policy-aware RDE with the help of wildcarded identity-based encryption(WIBE)[1] where key-pairs are generated using the policy as input to a function generating the keys[3][4].

Untrusted Scalable Storage

When a client creates an entity, the WAVE service places the entity's public key into the scalable untrusted storage. For attestations, it places the RDE ciphertext into the storage. As with RDE, the placement and retrieval of this data are transparent to the client since the WAVE service handles this.

This storage is decentralized: it is spread over multiple servers owned by different parties. The servers are only trusted to maintain availability, but not integrity or privacy. Integrity is enforced by a *Unequivocable Log Derived Map*(See section 2.6.4), which is an extension of a Certificate Transparency log and enables efficient proof of non-existence necessary for revocations. Privacy is achieved by RDE (section 2.6.3). Users and services can thus interact with any storage server without trusting the servers' operators, except for availability[4].

2.6.4 Unequivocable Log Derived Map

Since storage is decentralized and untrusted, there has to be a system to prevent dishonest parties from forging or removing attestations and revocations. One of the requirements of this storage is that it is append-only. A blockchain is a natural candidate for this. WAVE was originally implemented using the Ethereum blockchain. However, this solution was inadequate due to the fact that classical blockchains using proof-of-work are too slow when adding new attestations. They also lack an efficient way of proving non-existence which is needed for revocations[22]. Certificate Transparency solves the performance issue, but it also lacks proof of non-existence. Because of this, the people behind WAVE developed a transparency log they call an *Unequivocable Log Derived Map*(ULDM) which solves the issue of proving non-existence.[4].

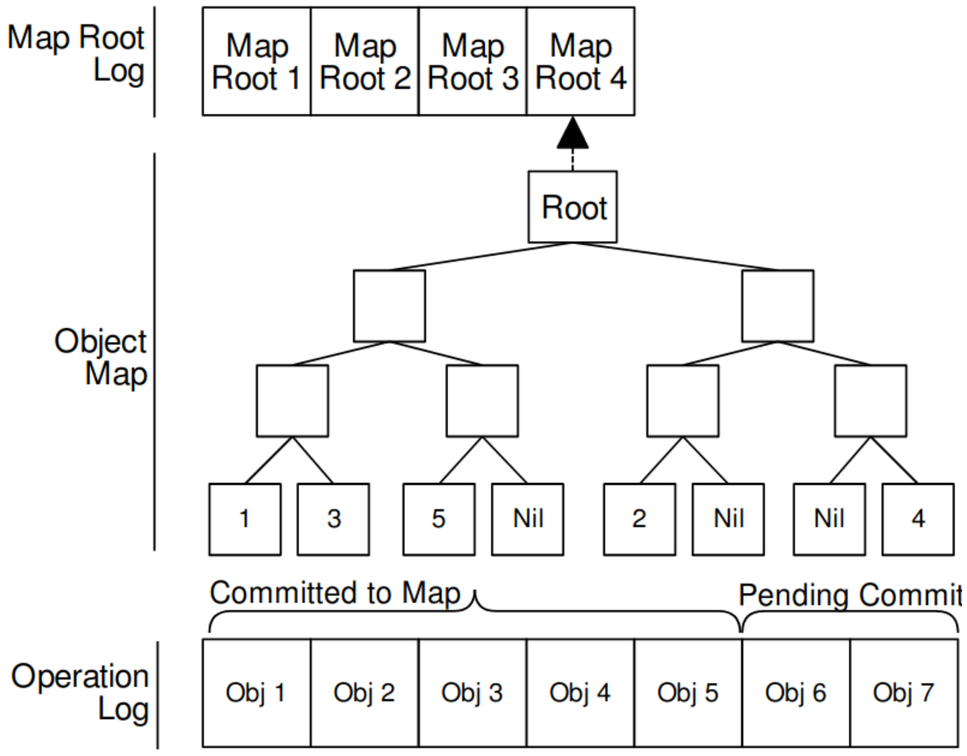


Figure 2.7: An Unequivocable Log Derived Map (ULDM) built from two Merkle tree logs and a Merkle tree map[4]

The ULDM consists of three Merkle trees. As shown in figure 2.7, the first tree is the Operation Log, which stores every insert operation. The Operation Log works just like a Certificate Transparency log[22][6], and is there to ensure that the ULDM is append-only. The second tree is the Object Map. The Object Map contains sorted hashes of the data added in the Operation Log. Since it is sorted it can be used to efficiently provide a proof of existence and non-existence. When something is added to the Object Map, the new Merkle tree root of the Object Map is added to the third and last Merkle tree, the Map Root Log. The Map Root Log is a blockchain of all the Object Map roots and is used for auditing (checking consistency between replicas of the ULDM).

2.7 Keycloak

Keycloak is an open source IAM solution aimed at modern applications and services. It makes it easy to secure applications and services with little to no code [2].

Users authenticate via a username and password, and in return get an authorization token that can be used to access protected resources. Resource servers (the server hosting the protected resource) will validate the token against the Keycloak server.

Unlike Firebase Authentication and Amazon Cognito, Keycloak is not a cloud based service. Keycloak servers can be hosted by anyone, and this removes

2.8 Keycloak vs. WAVE

Table 2.1 shows a comparison between WAVE and Keycloak based on the requirements specified in section 1.1.

Table 2.1: WAVE vs. Keycloak Comparison

	WAVE	Keycloak
Authentication	No	Yes
Authorization	Yes	Yes
Authorization Delegation	Yes	Yes
Decentralized Storage	Yes	No
Private Permissions	Yes	No
Offline Participants	Yes	Yes

Unlike WAVE, Keycloak is centralized. All user-data is stored on a centralized server. Also, permission policies are visible to the administrators of the Keycloak server, and they can view, edit or delete these permissions if they want.

In WAVE, permission policies they are stored encrypted in the ULDM storage which can be hosted by anyone, anywhere. Users only have to trust the ULDM storage providers for availability.

It is not possible for anyone to view these policies unless they are the issuer or subject of the policy. Modifying the ULDM is not possible without it being detected by auditors.

If a Keycloak server becomes unavailable, the entire system becomes unavailable. WAVE allows several replicas of the ULDM. These can be hosted at different locations. Consistency is ensured by auditors that check and compare the map root log.

An IAM system is not complete if it does not handle authentication. WAVE is an authorization framework, and does not handle authentication. However, if we build an authentication system on top of WAVE it can be a suitable decentralized alternative to Keycloak. Chapter 4 proposes an IAM design using WAVE for authorization.

/3 Method

This thesis will investigate an alternative to Keycloak for authentication and authorization in blind data miners. To find a suitable solution, I have to figure out if there are any problems that arise with Keycloak, and if so, look for alternatives in relevant literature that mitigates these issues. The most suitable solution will then be implemented and the implementation will be evaluated to see if they meet the requirements for use with blind data miners. Following Anne Håkonson's path on methods and methodologies[11], this thesis will conduct a *qualitative* research.

Since this is a known practical problem that needs to be solved, the research method that best fits in this situation is an *applied* method. The thesis builds on existing research on cryptography and privacy. Primarily cryptography[26][24] and privacy[31].

There will be an *abductive* approach to the problem. The design and implementation will be based peer-reviewed solutions for similar problems, and the implementation will be evaluated to see if it is a suitable alternative to Keycloak.

Carrying out research requires a research strategy. An *action research* strategy is chosen for this thesis. There are actions performed to contribute to the practical problem of authenticating and authorizing users that has data stored on blind data miners.

4 Design

This chapter will look at three designs for authentication and authorization in blind data miners(BDM). Two of which uses Keycloak, and one with WAVE for authorization with a custom public key protocol on top for authentication.

Chapter Outline

- **Section** 4.1 explains the chronic pain project and why BDMs are used.
- **Section** 4.2 covers how authentication and authorization are typically done today using tokens. This is a prerequisite for understanding the following section about Keycloak.
- **Section** 4.3 describes the current centralized IAM design used in the chronic pain project and why it might not be a suitable solution. Then, another Keycloak design that is semi-centralized is proposed in the same section. This second design aims to mitigate the problem with the currently adopted design, but also has some issues of its own.
- **Section** 4.4 describes a design that aims to mitigate the issues with centralized IAM systems. This design is built around the fully decentralized authorization framework WAVE, and is the core of this thesis.

4.1 Introduction

The idea behind the chronic pain project is to take patient-reported outcomes and experiences from treatments and use this data to make a tool that can be used to suggest treatments to patients with similar symptoms. The hope is that with enough data, it will be possible to see what kind of treatment works for different types of pain. So instead of trying to find a universal treatment for all chronic pain patients, the system used in the chronic pain project will hopefully be able to suggest personalized treatments based on information gathered from similar cases [5].

The designs described in this chapter are authentication and authorization solutions for the system with BDMs used in the chronic pain project. Since this is sensitive data, it is important that the data is not accessible to anyone other than those it is intended for. Data must be stored and protected according to data privacy laws such as GDPR[9] in Europe and HIPAA[13] in the US.

Patients are the data owners and should have full access to their own data, meaning they should be able to read, edit and delete data they have stored in the system. To give patients personalized treatments, healthcare professionals might have to look at their patients' data. Therefore, patients should be able to share data with others.

In the chronic pain project, patients will report their pain intensity and characteristics three times a day. This will be done with a mobile app. Patients will also be fitted with a Fitbit that automatically collects health metrics such as heart rate and sleep quality. This data is stored across three blind data miners(BDM), where each BDM is holding one share created by splitting the data using Shamir's secret sharing[26]. There are two main reasons for using BDMs. First, BDMs can be instructed by an analyst to privately compute statistics on all the data using their multiparty computation protocol described in [31]. The analyst can, in theory, be anyone since the protocol does not reveal anything about individual patients. The output is only aggregated statistics. However, the protocol should probably be limited to only a handful of researchers to not overload the BDMs with heavy computation tasks. Another reason for using BDMs is that it is far less likely that data will be stolen or accidentally leaked. Recall that BDMs are hosted at different independent organizations, so an attacker would have to breach all of them to steal the data. It is also unlikely that three independent organizations would accidentally leak data at the same time.

There needs to be an identity and access management system in place for all of this to work. There will be an assumption that the chronic pain project is on an invite-basis. It is only possible to register if you either get invited, or actively contact them to join the project.

4.2 Traditional authentication and authorization

The most common way of authentication is username/password authentication. Figure 4.1 shows how it is typically done. The user provides a password and unique username to an identity provider. In return the user gets an access token that can be used to access protected resources. Access tokens contain signed information about the user's permissions. User sends the token along with a request to a resource. Server validates the token with the identity provider, and fulfills the request if everything is fine.

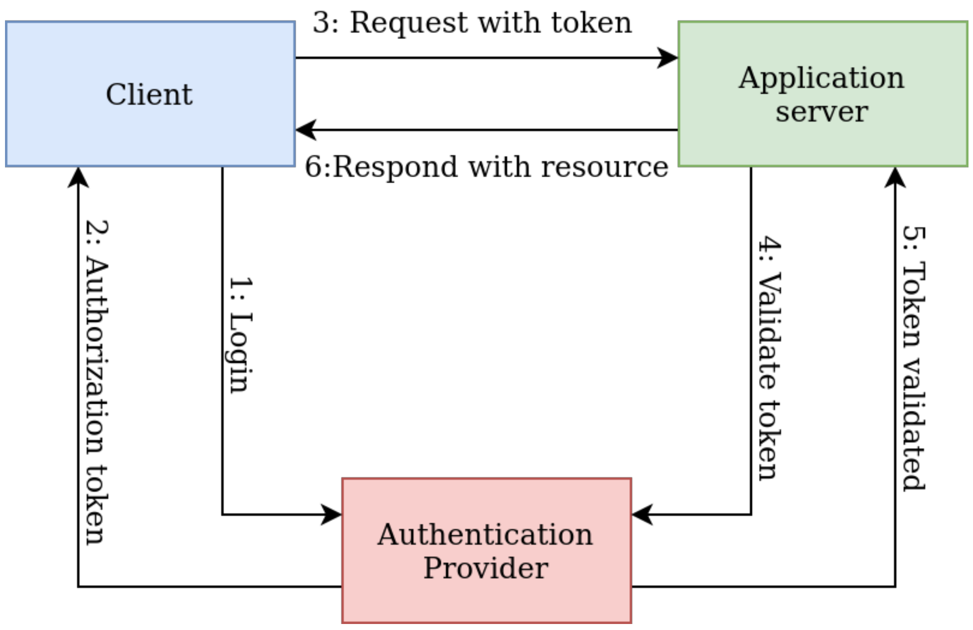


Figure 4.1: Token-based Authentication/Authorization

Application server and authentication server can be the same server. Using a traditional protocol with BDMs, the user would authenticate using a username and password, and get a token that it would send to all BDMs. Each BDM would then validate the token with the authentication provider. This is how it is done in the currently chosen solution in the chronic pain project. The authentication provider is a Keycloak server.

4.3 Keycloak

Keycloak is an open source token-based IAM solution. Keycloak is currently the chosen IAM solution in the chronic pain project. This section quickly covers two different designs using Keycloak with BDMs. The first design (section 4.3.1) is the one used in the chronic pain project. The second design (section 4.3.2) is a proposed design that aims to mitigate the problems with the first design.

Keycloak servers are independently run IAM servers. Authentication and token issuing are entirely separated from the application, so application developers do not have to deal with login forms, authenticating users, and storing users.

Communication between client, BDMs and Keycloak servers is done over HTTPS. Therefore, the Keycloak server and BDMs need to get a digital certificate signed by a trusted certificate authority(CA) like Google. Self-signed certificates is an option since the chronic pain project is not open to the public. If the project is open to the public it will be more difficult for users to trust the project since the majority of users will most likely never be in contact with the people behind the project. The mobile app can be shipped with a list of trusted self-signed certificates for Keycloak server and BDMs. If someone else try to create another self-signed certificate for those domains names or IP-addresses, they will not be in the list of trusted certificates and will therefore be rejected. However, obtaining a digital certificate signed by a trusted third-party is neither difficult nor expensive. There are even free options like Let's Encrypt[8]. CA-signed certificates is probably the best option, because it can, for example, prevent malware from injecting their own self-signed certificate to the list in the mobile app.

4.3.1 One Centralized Keycloak Server

This section describes the design for authentication and authorization that is currently used in the chronic pain project. It is comprised of a single Keycloak server issuing access tokens to the mobile app.

Figure 4.2 shows the architecture for this design. The BDM Data Integrator's job is to assemble shares to reconstruct the original data from the BDMs, which will be sent to the TSD (Tjeneste for Sensitive Data). TSD is a service for collecting and storing sensitive data[28]. This data can be accessed at the TSD or similar services by researchers.

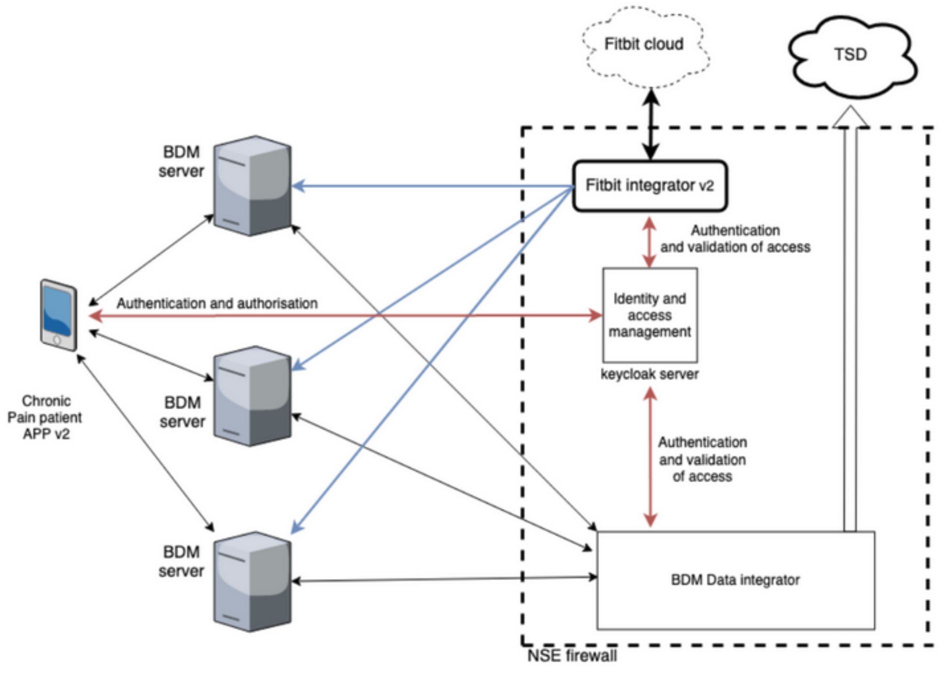

Fitbit cloud
TSD
BDM server
Fitbit integrator v2
Authentication and validation of access
Authentication and authorisation
Identity and access management
keycloak server
Chronic Pain patient APP v2
BDM server
Authentication and validation of access
BDM server
BDM Data integrator
NSE firewall

Figure 4.2: Architecture of the currently chosen solution at NSE[16]

When connecting to the BDMs, the mobile app is directed to the Keycloak server where the user logs in with his/her username and password. Upon successful login, the Keycloak server gives the client a token that is presented to the BDMs on each request. When BDMs receive a request with a token, they will check with the Keycloak server that the token is valid. The Fitbit integrator is responsible for getting the patients Fitbit-data from the Fitbit-cloud, and will put this data on the BDM after getting a valid token from the Keycloak server.

Why this might not be suitable with BDMs

Remember that one of the ideas of using BDMs is that it distributes the data so that leaking sensitive data is less likely. By using a traditional token-based authentication system with a centralized authentication server like the one described above, we introduce a vulnerable point in the system. An attack on the Keycloak server can compromise all users and their data.

4.3.2 Multiple Keycloak Servers

A possible mitigation for the problem that arises with a single Keycloak server is to have a Keycloak server for every BDM as illustrated in figure 4.3. This is a simpler figure where the BDM Data Integrator, Fitbit integrator, and TSD has been left out as it is not really relevant for understanding the difference between the two. The important thing to notice here is that every BDM is assigned their own Keycloak server, and will only accept tokens issued by that Keycloak server.

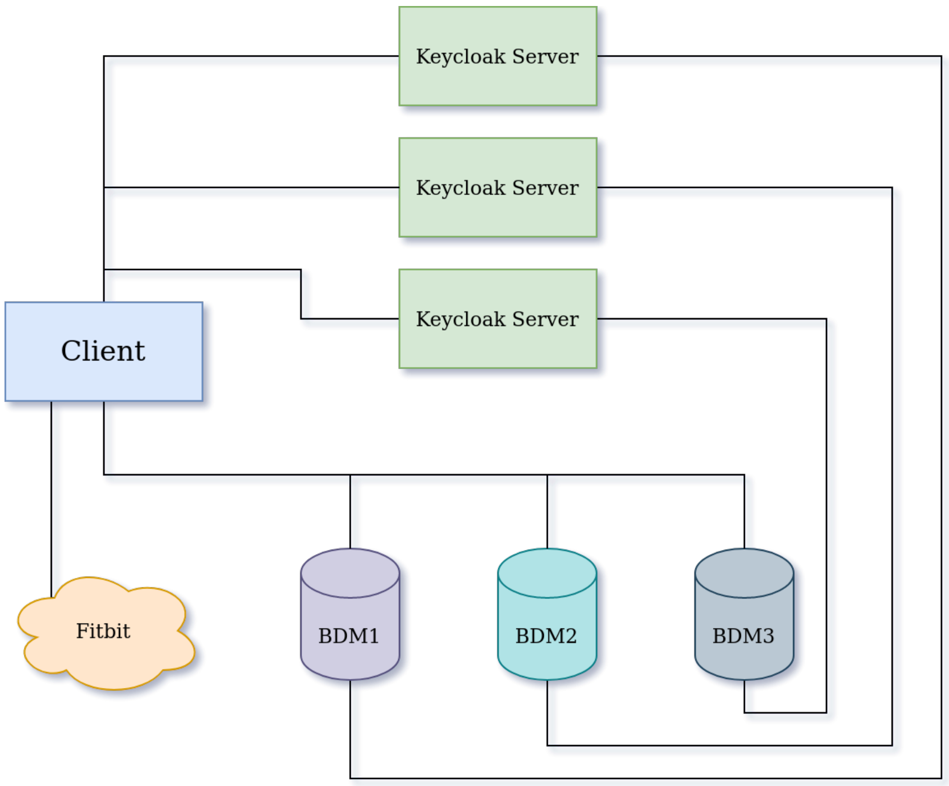


Figure 4.3: Proposed design with multiple Keycloak servers

The Keycloak servers should be hosted on different networks so that a breach in one of the networks does not affect the other servers. They could be hosted by the same organization that hosts the corresponding BDM server. Each BDM requires a different authorization token to respond to a request. Upon receiving a request, the BDMs validate the token with their Keycloak server.

If one of the Keycloak servers is compromised, it can lead to the leakage of every users' data from the corresponding BDM, but because of secret sharing, this data is useless without the shares stored in the other BDMs. A problem with this design is that if one of the servers go down, patients will not be able to access their data. There would have to be backup Keycloak servers to get around this issue.

Another problem with this design is that the user need to authenticate with multiple Keycloak servers. The user would have to have a different password for every one of them, or else the leakage of passwords from one of the servers would compromise all of them.

4.4 WAVE

This section describes a design for authentication and authorization that is built around the fully distributed authorization framework WAVE. WAVE does not rely on a traditional centralized token-based scheme. The goal of this design is to mitigate the problems from both of the Keycloak designs described in section 4.3

Section outline:

- **4.4.1** is a quick recap of WAVE which is described in more detail in section 2.6.
- **4.4.2** describes a protocol for authentication that will be used together with WAVE authorization. It also covers user registration.
- **4.4.3** describes how authenticated users are authorized to access resources on the BDMs using WAVE. The section also describes how a request is done from start to finish.
- **4.4.4** describes how data access is shared with other users.

4.4.1 WAVE Recap

WAVE is an authorization framework offering decentralized trust. There is no centralized authority that handles permissions. Everything is stored in publicly auditable storage known as a *Unequivocable Log Derived Map* (ULDM) that, with the help of Merkle trees, can not be modified without anyone noticing[4]. This means that you do not need to trust the storage provider because if it misbehaves it will be detected and can be permanently removed as a storage provider. Like certificate transparancy logs, it is possible to have several independent

ULDM storage providers, and the consistency between them are guaranteed by auditors that compare the map root hash (see Figure 2.7)

Terminology

A quick recap on WAVE terminology

- **Entity**. An entity in WAVE are bundles of public/private keypairs and correspond to a user, server or group.
- **Authorization Graph**. WAVE is a graph-based authorization framework. nodes represent entities and edges represent permission grants from one entity to another.
- **Entity hash**. A unique hash that is used for identifying entities in WAVE.
- **Attestation**. Attestations are signed certificates that contain a description of permissions that has been granted to an entity.
- **Proof**. A proof is a signed concatenation of decrypted attestations. It is used to prove access to resources.
- **WAVE Service**. The WAVE service makes attestations, builds proofs, verifies proofs, and handles communication with the ULDM(Unequivocable Log Derived Map) storage. It is a background process that runs on each device.

Entities in the chronic pain project are BDMs, patients, administrators, health-care workers and researchers.

RTree Policy and Namespaces

An RTree policy manages permissions on a hierarchically organized set of resources. Resources are denoted by a URI pattern such as *BDM1/entity-hash/pain-level*. The first element of a URI (e.g.BDM1) is called the namespace authority or just namespace, which specifies the entity who is the root of authorization for that resource (the entity who has permission on that policy without having received permission from someone else).

4.4.2 Authentication

Authentication is typically done using a username and password. Since WAVE is a framework for distributed authorization, it is natural to go for an authentication scheme that does not rely on a centralized authentication server.

When setting up a secure communication channel, both sides must know with whom they are communicating, so they do not send sensitive data to someone they did not intend to send it to. Therefore, the BDMs need to prove their identity to the user, and the user must prove its identity to the BDMs.

This section proposes a solution that uses public key authentication for authenticating both servers and clients. While regular X.509 digital certificates are useful for server authentication, it is not feasible to expect every user to obtain their own certificate signed by a trusted certificate authority (CA) since it requires a bit of effort to get one. It needs to be an easy and intuitive process for users with no technical background. Therefore, client authentication is done slightly different from server authentication.

BDM Authentication

Secure connection and server authentication are done using SSL/TLS(Secure Sockets Layer/Transport Layer Security), which are protocols for secure communication over the internet. The BDMs have a digital certificate that contains their public key and additional information about who they are. The authenticity of these certificates can be vouched for by a trusted third-party certificate authority, or they can be self-signed.

As discussed in the Keycloak-solution (section 4.3), it should be enough to ship the app with a list of trusted self-signed certificates for BDM authentication, but again, obtaining a certificate signed by a trusted CA is neither difficult nor expensive.

BDM Administrator

To explain client authentication, we must first know what kind of data is involved, how it is generated, and how and where it is stored. The client authenticates with a public key protocol. Client sign some data with their private key, which can only be verified with the corresponding public key. This proves that a client is in possession of certain private key. To avoid usernames and passwords, there has to be an administrator that handles user registration. For example, when setting up public key authentication on remote servers via

SSH, you have to log in with a username and password before registering your public key. BDM administrators will be able to upload public keys and entity hashes to the BDMs.

When a BDM server is set up at an organization, it must be initialized with some entities that act as administrators. Administrators are given special permissions. Most notably, the ability to register users. They are not given access to anyone's data. Data permissions can only be delegated by the data owner.

User Registration

While SSL/TLS authentication could work for authenticating clients, it was deemed easier to avoid setting up digital certificates for every client.

To avoid a lot of garbage data in the BDMs, the system should probably not be open to everyone. To restrict access, user registration will be done through an administrator.

When someone decide to use this system, the user downloads the app. During initial setup, the app generates an RSA key pair used for authentication and a WAVE entity that will represent the user. Recall that WAVE entities are bundles of private/public keypairs. These keys are only used for authorization and are not part of the authentication system. The app then generates a QR-code containing the entity hash and public key. The administrator uses his app to scan the QR-code and sends this data to each BDM for registration. At the same time, the system generates the WAVE-attestations, which gives the new user access to their own data, and the possibility to delegate this access to others. This process will be described in a later section. The corresponding private key is stored securely on the patients' phone, encrypted with a password chosen by the patient during the initial setup. The entity hash is used as a unique identifier in the system.

Data needed for authentication

BDMs must store the data used for authentication without splitting them into secret shares, so this data can not contain any sensitive information. The only thing needed is the entities' authentication public keys and their WAVE entity hash. It does not matter if this data leaks or is stolen because both key and hash are public anyways, and is useless if you do not have the corresponding private keys.

Client Authentication Flow

This section explains how authentication is done on *one* BDM server. The process is identical on all servers.

First, the server must authenticate itself for the client and set up a secure connection. This is done using SSL/TLS with either a CA-signed or self-signed certificate.

Next, the client must provide its identity to the server. This process is loosely based on SSH and SSL/TLS handshakes. It is a simple public key authentication protocol that is only intended to be used in this specific environment *after* a SSL/TLS handshake has been executed successfully. The protocol works as follows (See also figure 4.4):

1. The client sends an authentication message with its entity hash and public key.
2. Server checks if a user with this hash and public key exist. If not, authentication fails.
3. Server sends a random string to the client.
4. Client uses its private key to sign the random string and sends this back to the server.
5. Server verifies the signature using the client's public key, proving that the client is in possession of the corresponding private key.
6. Client is authenticated and can send requests to authorized services.

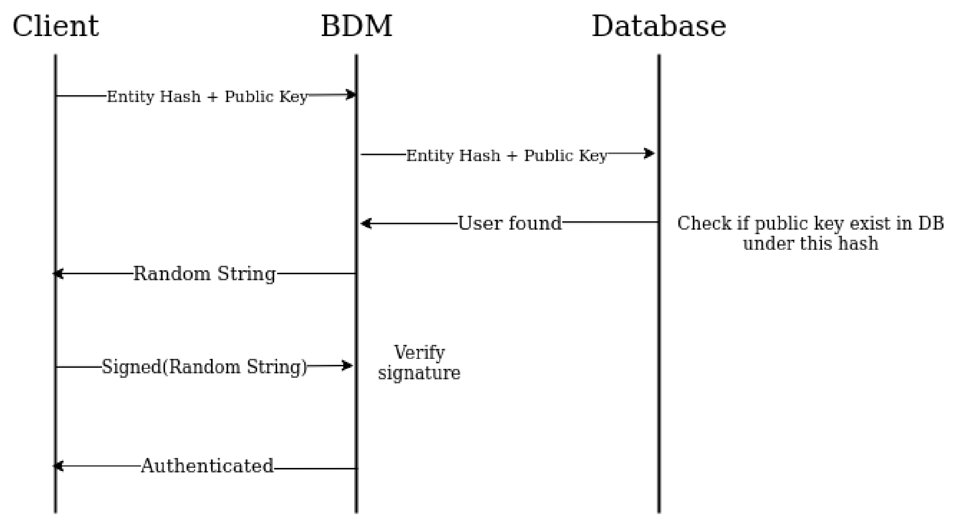


Figure 4.4: Public key authentication flow

4.4.3 Authorization

Authorization is done using WAVE. WAVE is a decentralized authorization framework. The decentralization is the main reason why WAVE was chosen. There is no centralized storage that can be attacked and compromise all users. WAVE also offers transitive delegation which is useful when a patient wants to share access to their data with others. Another property that WAVE offers is the ability to delegate permissions to offline participants. While this is trivial to do in centralized authorization, other decentralized solutions (e.g. SDSI/SPKI[25][32] and Macaroons[7]) does not offer this property. We can not expect patients or healthcare workers to be online at all times.

Namespace Authority

Each BDM is the authority entity in their own namespace. They will have access to all data. Recall that each BDM only holds one share from a secret sharing split, so a single BDM can not see any personal information about its users.

When an entity is registered in a BDM by an administrator, the BDM will create an attestation that gives this entity full access(read, write, edit, delete) to all data stored under their entity hash. They will also be given permission to share their access rights with other entities.

Patients can access their data from BDM1 using the following RTree policy

URIs: *BDM1/entity-hash/**. This will give the patient the share from BDM1. To get all shares a request must be sent to all BDMs to the same URI except that the namespace authority (which BDM server) must be different. Figure 4.5 shows how each BDM is the authority of their own namespace illustrated by being the first nodes in the graph, and they have all granted the patient access to their data.

The figure only shows *read* rights, but in reality they have full access.

Requesting Resource

Here we will describe the entire process for retrieving and displaying patient data stored on BDMs. All of the technical bits are transparently handled by the app, WAVE client and BDMs.

Requests are done via the patient's mobile app. First, the client must decrypt the private key stored on their device used for authentication on the BDMs (this key was created when the user first registered as described in section 4.4.2). To do this, the patient has to enter a password or use biometrics like a fingerprint. The password/biometric data is the secret that is used to decrypt the key. There is no network connections involved in this step. Everything happens locally on the patient's device and the purpose is only to decrypt the private key.

Next, the app connects to all BDMs, which authenticate themselves using their digital certificates and at the same time exchange encryption keys for secure communication. Next, the patient authenticate using their public key as described in section 4.4.2. After this, the patient has established a secure and authenticated connection to all BDMs. The next step is to provide something to the BDMs that proves the patient is allowed to access the requested data. This is done by using the WAVE service to create what they call a *proof*, which will be sent to the BDMs for them to validate.

To explain how this proof is made, we must first look at the relevant portion of the WAVE authorization graph. Figure 4.5 shows this. The BDMs has previously (section 4.4.2) created attestations containing policies that gives the patient access to all data that is stored under the patient's entity hash. The attestations are represented as the edges between the BDMs and patient node. The different colored padlocks means that the attestation is encrypted using a public key of key-pairs that are generated specifically for this attestation using wildcarded identity-based encryption(WIBE). The patient has the corresponding private key to decrypt these attestations to form proofs. WIBE is necessary for the reverse-discoverable encryption (RDE) to be policy aware as discussed in section 2.6.3.

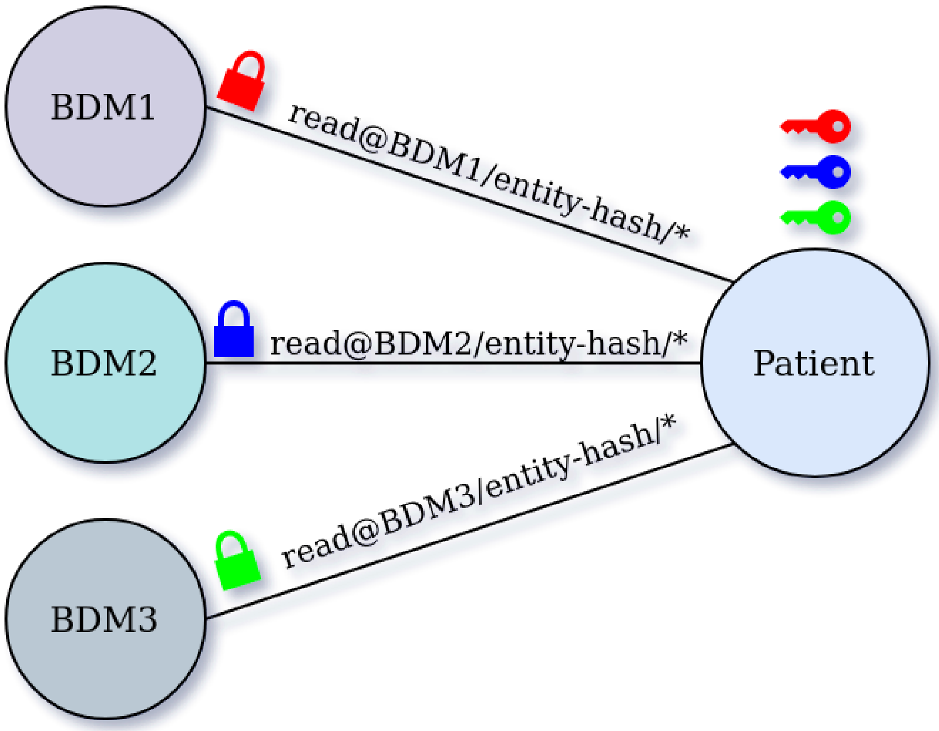


Figure 4.5: WAVE Graph with a single patient

RDE is used when access rights has been shared with someone else as described in section 2.6.3. This will become more clear later when data sharing is described.

The patient's app generates the proof using the WAVE service, which decrypts the attestations between BDM and patient using the WIBE private key. Decrypted attestation is illustrated with colored padlocks, and the colored keys illustrate the private keys for decrypting them. Decrypted attestations are concatenated and signed by the proving entity, which in this case is the patient that want to prove that it has access to the resource. The WAVE service handles this. These signed, concatenated attestations are the *proof* and can be verified by anyone.

Requests are done using a protocol specifically designed for this environment. Request messages are JSON objects with the request type, the URI for the requested resource, and the WAVE proof file that act as an access token except that it was generated without the use of a centralized authorization server. This JSON object is sent to every BDM.

The BDMs receiving the request uses the WAVE service to verify that the provided proof is real, belongs to the authenticated user(entity hash), and has not been tampered with. If successful, the WAVE service outputs the information shown below

- **Referenced attestations:** A list with the hash of all attestations that are used to form the proof. The hashes are given a number which is used in for the paths field.
- **Paths:** The attestation path from the granting entity to the receiving entity.
- **Subject:** The entity hash of the subject that this proof belongs to.
- **Expires:** The date when these permissions expires.
- **Permissions:** The permissions that this proof proves. e.g read, write, delete.
- **URI:** The resource that the subject has access to with this proof.

If everything is valid, the BDM responds with the requested resource. In this case the shares that are stored under the request URI. All steps described above are illustrated in Figure 4.6.

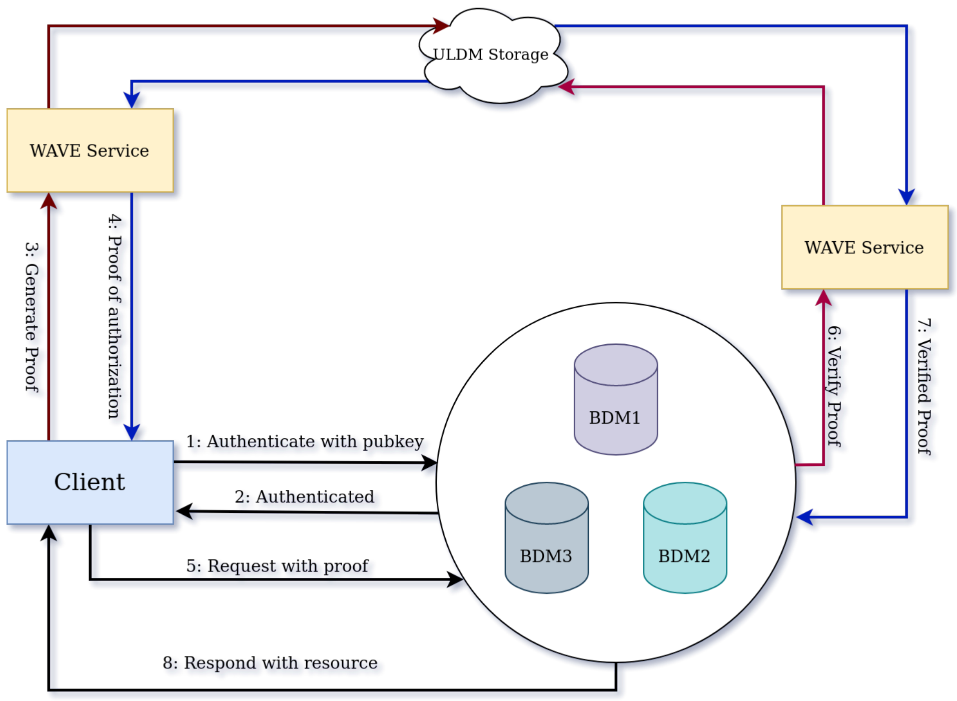


Figure 4.6: BDM Request Flow

Reconstruct data

The patient's app request data from all BDMs simultaneously and then reconstructs the original data using lagrange interpolation. This happens on the patient's device, so there is never fully reconstructed sensitive data being transmitted over the network. After reconstruction, the data can be displayed to the patient.

4.4.4 Authorization Sharing

An entity can share their access rights with other entities. In the chronic pain project it will typically be patients sharing their data with their doctor. WAVE allows the granting entity to specify how many re-delegations is allowed.

To share data with someone else, they need the recipient entity's hash. There are many ways this can be done, all with their own pros and cons. For now, assume that entities meet physically and share hash via QR-code like it is done in user registration discussed in section 4.4.2. The granting entity scans the QR-code with their app, selects which data they want to share and the type of access they share (e.g. read, write, delete). The app will then use the WAVE service to create attestations and updates the global authorization graph with the newly granted permissions. Figure 4.7 shows how the authorization graph looks when access has been shared. It is the same as before (Figure 4.5) between BDMs and patient. When patient shares access through the WAVE service, it creates attestations that are encrypted with new WIBE keys generated for the doctor. Included in the attestations are the keys necessary to decrypt upstream attestations. In this case, it is the keys that the doctor need to decrypt attestations between BDMs and patient to form a proof of authorization when accessing the patient's data.

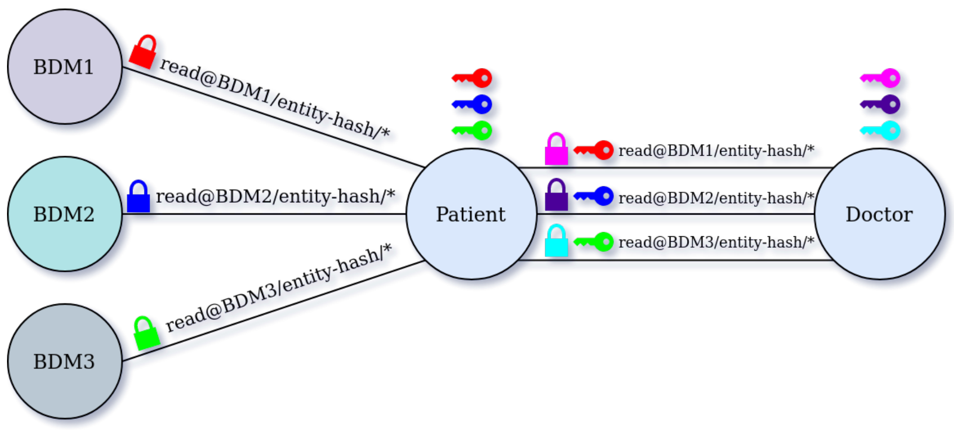


Figure 4.7: WAVE graph with delegation

Figure 4.8 shows two patients that has shared data with their doctor. This is where the reverse-discoverable encryption mentioned previously in section 4.4.3 using WIBE keys is necessary. Doctor 2 should not and can not decrypt any attestations on the path from BDMs through patient 1 because he/she does not possess the necessary keys to do so, and can therefore not form a proof of authorization to patient 1's data. WIBE generated keys are necessary for policy aware encryption. Remember that patients also has write-access to their data. If the RDE used keys generated based only on the patient ID, the key included in the attestation between patient and doctor could also be used to discover write-access policies. The keys are therefore generated using the patient ID *and* policy.

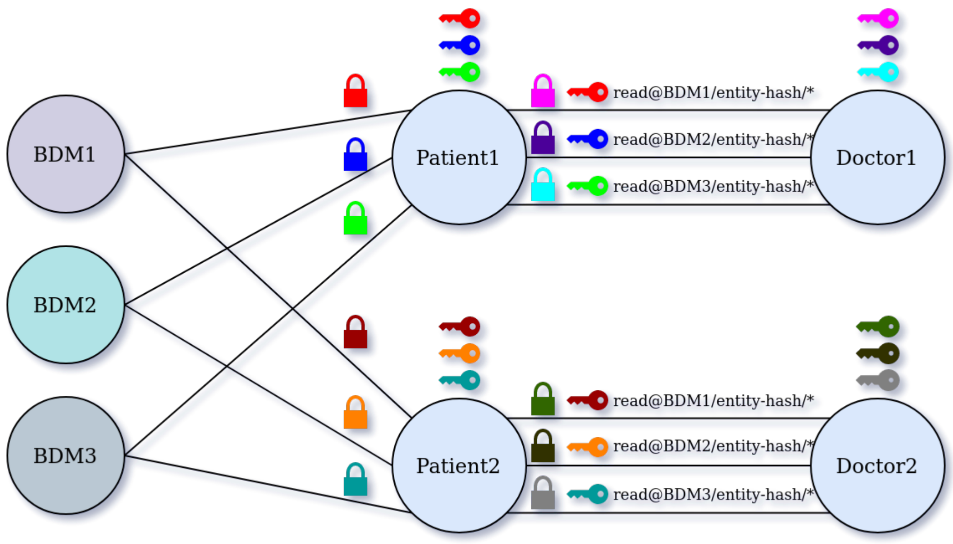


Figure 4.8: WAVE graph with two patients and delegation

Finally, a doctor might have to share access with someone else (e.g. specialists). If the patient has given the doctor permission to delegate access, the doctor can do so in the same way as the patient did using the app and QR codes. Figure 4.9 shows the WAVE graph after this. There are still three attestations between the entities, but has been compressed to one line in this figure to make it cleaner.

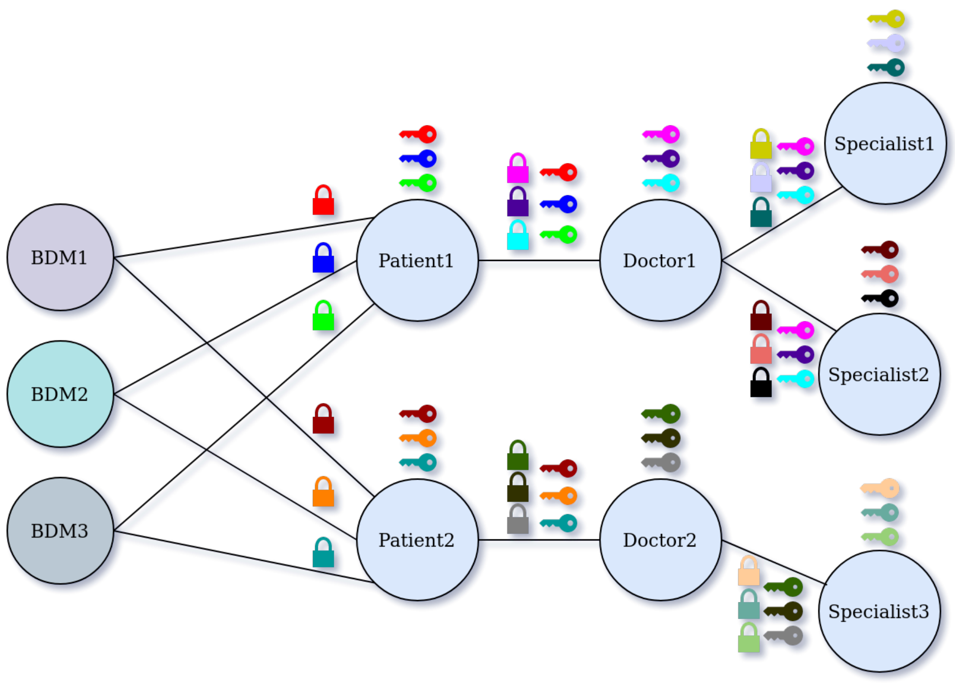


Figure 4.9: WAVE graph with several indirections

When specialist1 creates a proof of authorization to patient1's data, it is done by decrypting the attestations on the path from specialist1 to the namespace authorities (in this case BDM1, BDM2, and BDM3). In Figure 4.9 you can see that specialist1 is able to discover all keys (through RDE) needed to do this, but specialist3 can not create a proof for patient1's data because the keys discovered will not be able to decrypt those attestations.

5 Implementation

To demonstrate the capabilitiess of WAVE, the design described in section 4.4 has been implemented. This chapter describes this implementation. A simple command-line interface (CLI) has also been implemented to interact with the system. The design describes a mobile application, but that application has not been implemented. However, the implementation described here is independent of the method of user input (e.g a mobile app). Reconstruction of secret shares has also not been implemented because that is not important part in an identity and access management (IAM) system.

The blind data miner(BDM) side of the implementation only handles authentication and authorization and does not implement any of the BDMs' core functionality (e.g. secure statistical computation). When a client request to read-/write/delete/modify data on a BDM, the BDM will only call mock-methods for doing so. Data is accessed using URIs, and the implementation handles granting and checking access to URIs, but exactly how data should be stored is outside the scope of this thesis.

Chapter Outline

- **Section 5.1** covers the choice of language and libraries used in the implementation.
- **Section 5.2** describes how communication between BDM and client is

implemented.

- **Section 5.3** describes how the authentication protocol in figure 4.4 is implemented.
- **Section 5.4** describes how authorization is implemented using WAVE.
- **Section 5.5** describes the BDMClient-class which is responsible for communicating with a single BDM.
- **Section 5.6** describes the implementation of BDM servers.
- **Section 5.7** describes the CLI Client which is placeholder for the mobile app described in chapter 4. This is what the user uses to interact with the system.

5.1 Language and libraries

The original BDM implementation is done in Java, and this implementation also started out as a Java implementation, but Java proved to be very complex when dealing with SSL networking and especially confusing with RSA.

Therefore this implementation is done in Python. Another reason for why Python was chosen is because it is a very high-level language which makes it easy to quickly implement a proof-of-concept in a short amount of time.

These are the libraries used which are worth mentioning.

- **json**[1] - A JSON encoder and decoder that converts dictionaries to JSON strings and vica versa. Request to BDMs are done using JSON objects.
- **PyCryptodome**[2] - A library for low-level cryptographic primitives. Used for digital signatures and handling of private/public keys.
- **sqlite3**[3] - A lightweight disk-based database that does not require a separate server process. Used for storing users.

1. https://docs.python.org/3/library/json.html
2. https://www.pycryptodome.org/en/latest/
3. https://docs.python.org/3/library/sqlite3.html

- **ssl**[4] and **socket**[5] - For networking and access to TLS/SSL.
- **subprocess**[6] - A library for starting new processes. Used to start and interact with the WAVE CLI.

5.2 Communication Protocol

A REST API would have been a natural candidate for communication between BDM and client. However, REST and HTTP is stateless, and this implementation is dependent on a stateful protocol. FTP was considered, but it was difficult to do authentication with FTP. Therefore, communication is done over plain TCP sockets with a custom JSON-structure. To avoid having to authenticate between every request, the socket connection is kept open and state (mainly who the user on the other side of the socket is) is preserved between requests. The BDMs are not designed with web browsers in mind, so it has not been considered important to use protocols that support web browsers.

5.3 Authentication

5.3.1 BDM Authentication

BDM authentication is done using SSL. Each BDM has its own digital certificate. In this implementation, they are self-signed. The client has these certificates in a list of trusted self-signed certificates. If someone else tries to authenticate with another self-signed certificate for those domain names, they will be rejected. Using Python's SSL-library that uses OpenSSL[7], BDM authentication is achieved, and an encrypted communication channel is established between client and BDM.

The code below shows how this is done on the server(BDM) side. `self.context.load_cert_chain()` takes as argument the server's digital certificate, the server's private key, and the password for the private key since it is encrypted on disk.

```
self.context = ssl.SSLContext(ssl.CERT_REQUIRED)
self.context.load_cert_chain(certfile=certificate, keyfile=key,
    password=pw)
```

4. https://docs.python.org/3/library/ssl.html
5. https://docs.python.org/3/library/socket.html
6. https://docs.python.org/3/library/subprocess.html
7. https://www.openssl.org/

```

self.serversock = socket.socket()
self.serversock.bind((host, port))

newsocket, fromaddr = self.serversock.accept()
sslsoc = self.context.wrap_socket(newsocket, server_side=True)
```

Client side is shown below. `self.context.load_verify_locations()` takes as argument a path to all trusted server certificates. `verify_mode=ssl.CERT_REQUIRED` makes sure that if the other side of the socket does not send a certificate, the connection will be terminated immediately. On both sides, when the socket is passed to `self.context.wrap_socket()`, the SSL handshake takes place and a new SSL socket is returned. The original non-SSL socket is closed.

```
self.context = ssl.create_default_context()
self.context.verify_mode = ssl.CERT_REQUIRED
self.context.load_verify_locations(capath="client/
    trusted_certificates")

self.sock = socket.create_connection((self.HOST, self.PORT))
self.ssock = self.context.wrap_socket(self.sock,
    server_hostname=self.HOST)
```

5.3.2 Client authentication

SSL can also do client authentication if clients have a certificate of their own. However, a custom authentication algorithm has been implemented for client authentication. It is very similar to regular public key authentication with digital certificates. It uses a digital signature to prove that the client is in possession of a private key that corresponds to a known public key tied to a unique username. The username is the entity hash of the WAVE entity belonging to a user (patient, health care worker, BDM, or administrator). Entity hashes are stored in each BDM's database together with the entities' public key.

The client sends an auth message over the SSL socket created previously, containing its entity hash and public key. The BDM will then check its database to see if this entity hash and public key are registered. If it does, the BDM will return the same message but with an added, randomly generated string. The client uses its private key to sign this returned message. The signature is sent to the BDM, which verifies the signature with the public key stored in the database for that entity's hash. The verification checks that decrypting the message with the public key yields the message sent previously with the random string. If it does, it means that the client on the other side of the socket has the corresponding private key.

Code below is the server-side of client authentication. Signing and signature verification is handled by the RSA module from the Python-library PyCryptodome.

```python
def authenticate_client(cli_sock, db):
    try:
        auth_msg = json.loads(recv_all(cli_sock))

        username = auth_msg["username"]
        public_key = auth_msg["pubkey"]

        retrieved_pubkey = retrieve_user_pubkey(username, db)
    except JSONDecodeError:
        raise AuthenticationError("Broken auth message")
    except IndexError:
        raise AuthenticationError("User not found")

    if public_key == retrieved_pubkey:
        response_msg = {
            "username": username,
            "pubkey": public_key,
            "server_random": str(uuid.uuid4())
        }

        resp_msg_encoded = json.dumps(response_msg).encode()
        send_all(cli_sock, resp_msg_encoded)

        signature = recv_all(cli_sock)

        response_msg = json.dumps(response_msg)
        pub_key = get_public_key_from_string(retrieved_pubkey)

        if(verify(response_msg, signature, pub_key)):
            resp = {
                "status": "OK"
            }

            send_all(cli_sock, json.dumps(resp).encode())
            return username
        else:
            resp = {
                "status": "FAILED",
                "error_msg": "Authentication failed! Invalid signature."
            }

            send_all(cli_sock, json.dumps(resp).decode())
            raise AuthenticationError("Client provided invalid signature")

    raise AuthenticationError("Public Key does not match stored key")
```

Client-side of the authentication protocol is shown below.

```
def authenticate(server_sock, entity_hash, priv_key):
    pub_key = get_public_key(priv_key).decode()
    auth_msg = {
        "entity_hash": entity_hash,
        "pubkey": pub_key
    }

    send_all(server_sock, json.dumps(auth_msg).encode())

    resp = recv_all(server_sock).decode()
    signature = sign(resp, priv_key)

    send_all(server_sock, signature)

    resp = recv_all(server_sock)
    resp = json.loads(resp)

    if resp["status"] == 'OK':
        return
    else:
        error = resp["error_msg"]
        raise AuthenticationError(f"Authentication failed.
   Error: {error}")
```

5.4 Authorization

Authorization is done using WAVE. In the current WAVE version, the interaction with the WAVE service is done with a command-line interface (CLI). To use this CLI, the user needs to know WAVE, but we can not expect users of this system to possess this knowledge. The client implemented is also a CLI, but a more user-friendly CLI that is an abstraction layer that hides the technical bits of the WAVE CLI that is not important for the user to know. This thesis does not focus on the visual design of the mobile app, but on a protocol for authentication and authorization.

The front-end CLI will run the WAVE CLI using Python's `subprocess` library.

5.4.1 WAVE CLI

The most important features of the WAVE CLI is creating an entity, granting permissions, creating proofs, and verifying proofs.

`./wv mke` will create a new entity. The CLI will ask for a password that is used to encrypt the keys that belong to this entity. The entity hash for the newly created entity is returned.

`./wv rtgrant --subject <entity_hash> --attester <entity_hash> --indirections <number> <permission_set>:<permissions>@<namespace>/<uri>` will grant permissions to the entity under `--subject`. The `--attester` entity hash is the one giving the permissions, and must provide its password to be able to grant these permissions. `--indirections` says how many redelegations is allowed. `<permission_set>` identifies the meaning of a set of permission strings. This is so that one person's idea of what read means does not accidentally get confused with another person's idea. `<namespace>` and `<uri>` is the resource that is being granted access to under the provided namespace

`./wv rtprove --subject <entity_hash> <permission_set>:<permissions>@<namespace>/<uri>` generates a signed proof that the subject has permissions to the provided resource. The subject's password must be provided to generate this proof.

Finally, `./wv verify <proof>` verifies that the provided proof is valid. The output is

The WAVE CLI can also be used to revoke permissions, but this functionality has not been implemented in the front-end CLI.

5.4.2 WAVE Module

A Python module that runs the WAVE CLI as a subprocess has been implemented. It contains a set of functions that hides some of the complexity of running the WAVE CLI. The module implements the functionality to create new entities, grant permissions, generate proofs, verifying proofs, and parsing a proof to easier check what the proof is actually proving.

```python
def generate_proof(entity_hash, permission, uri, pw)
def grant_permission(issuer, subject, policy, pw)
def create_entity(pw)
def verify_proof(proof, namespace, entity_hash)
def parse_proof(proof)
```

These functions run the commands described above in section 5.4.1. We can take a closer look at `grant_permission()` to see how it is implemented.

```python
def grant_permission(issuer, subject, policy, pw):
    permission = policy["permission"]
    indirections = policy["indirections"]
    URI = policy["uri"]

    p = subprocess.Popen(["./wv", "rtgrant", "--subject",
   subject, "--attester", issuer, "--indirections",
   indirections,f"patientdata:{permission}@{URI}"], stdin=
   subprocess.PIPE)

    pw = pw + '\n'
    p.communicate(pw.encode())
```

The function takes as input the entity hash of the one issuing permissions, the entity hash of the subject receieving permissions, a policy that contains permissions, indirections, and URI to a resource, and finally the password for the entity issuing the permissions. As we can see, simply running this function is a lot easier than having to manually run the WAVE CLI every time.

The implementation is similar for the other functions. They run the WAVE CLI as a subprocess with the required input.

5.5 BDMClient

BDMClient is a class that is responsible for talking to a single BDM. When a BDMClient-object is created, it takes as argument a BDM IP address and port number, the namespace for the BDM, the WAVE entity hash that the client represents, and the private authentication key for this entity.

During initialization, the BDMClient loads a list of trusted BDM certificates from the devices' filesystem and sets up a SSL context. There is a separate method for connecting to the BDM, so the one who creates the object must explicitly call `connect()` to connect. `connect()` will execute the BDM authentication procedure described in section 5.3.1. Next, client will initiate the client authentication procedure described in section 5.3.2. If everything succeeds, a secure and authenticated socket connection is kept open until the `disconnect()` method is called.

At this point, the BDMClient can send requests through the socket with a set of methods listed below.

```
def store_data(self, data, uri, password)
def read_data(self, uri, password)
def delete_data(self, uri, password)
def modify_data(self, uri, data, password)
```

These methods will run the WAVE module functions to generate a proof for the requested URI (read-permission for `store_data()` etc.)

Requests are JSON-objects that contain the information necessary for the server to handle the request. Below are examples of a read and a write request.

```
{
    "type": "read",
    "uri": "<entity_hash>/pain_level/30-05-2021",
    "proof": <proof>
}

{
    "type": "write",
    "data": <data>,
    "uri": "<entity_hash>/pain_level/30-05-2021",
    "proof": <proof>
}
```

`<proof>` is the bytes from a proof that proves access to the URI. The proof, of course, also contains the subject that has this access. The server holds the connection open and keeps the state between request, so it already knows and has authenticated the subject that sends these request. `<data>` is the data that should be stored in a write or modify request

5.6 BDM Server

BDM servers are also implemented as a class. Like with BDMClients, the initialization takes a set of arguments. These are the BDM entity hash, hostname and port number, its digital certificate and the private key that belongs to the certificate, and a password that is used to decrypt this private key and the password used for the WAVE service. Initialization will set up a SSL context, create a TCP-socket and bind it to the hostname and port. This is the socket that listen for incoming connections. This class has a `run()` method that starts listening for connections. When a client connects, a new socket for that client is created and wrapped in the SSL context. Then the SSL handshake happens and the BDM authenticates itself for the client using its digital certificate. This creates a secure connection between client and BDM. Now the client can authenticate using the procedure described in section 5.3.2. If authentication succeeds, the BDM server will wait for a request from the client. The socket is kept open until the client disconnects.

It is implemented so that it should be easy to start several BDMs. A config-file is used to configure the BDM

```
{
    "bdm_id": "BDM1",
    "hostname": "localhost",
    "port": 5050,
    "cert": "bdm1.pem",
    "key": "bdm1_key.pem",
    "db": "bdm1_db.db",
    "key_pw": "bdm1"
}
```

Key password should probably be stored as an environment variable or entered manually when the BDM is started. For now, it is stored in the config-file.

5.6.1 Handling requests

A request is received over the SSL socket. The first thing that happens is that the BDM will use the `verify_proof()` method from the WAVE module that runs the WAVE CLI to verify that the proof is valid and that it belongs to the authenticated user. It will also check that the proof belongs to the namespace that the receiving BDM is the namespace authority for. If something is wrong, a response with an error message is returned to the client, and the connection is terminated.

If the proof is successfully verified, the next step is to check the contents of the proof. BDM will check that request type (e.g. read) is in the proof's list of permissions. and that the URI requested is in the proof. Proofs can contain wildcarded URIs. For example, a request can be sent to the URI shown below

```
"uri": "<entity_hash>/pain_level/30-05-2021"
```

and the proof URI can be as the one shown below. This would be a valid proof for the requested URI.

```
"uri": "<entity_hash>/*"
```

Typically a patient would be granted all types of permission to the URI shown above because they should have full access to all data they store. A patient can chose to share access to only the data stored under `pain_level`. They would create an attestation encrypted with the receiving entity's public key that contains the following URI.

```
"uri": "<entity_hash>/pain_level/*"
```

If the user is authorized to do the request, the BDM will call mock-methods for reading, writing, deleting or modifying data. Nothing will be stored or read.

As mentioned before, what happens next is outside the scope of the thesis. An OK message is sent back to the client saying that the request was successfully fulfilled

5.7 CLI Client

The CLI client is a placeholder for the mobile app in this implementation. As discussed earlier, the method of input should not matter for the core of the implementation.

The CLI client's main job is to create and manage all BDMClient objects. The CLI client will, on startup, create a BDMClient object for every BDM that is part of the system and run their `connect()` method, which will open up a connection to the BDMs and run the authentication protocol once for every BDM.

Before the objects can be created, the user will be prompted to enter a username, which will be used to find a config-file containing the user's entity hash and the alias for the public and private key used for client authentication. The alias is just the filename for the keys. The user must also provide a password to decrypt the keys. The decrypted private key, the password, and the entity hash are needed when creating the BDMClient objects.

If a config file for this username does not exist, the client will run a procedure for creating a new user. First, a username must be chosen. This username will be used to find the config file and will also be the alias for the keys. Then the user must select a password that will be used to encrypt the public and private key. After a username and password have been chosen, the client will first generate a key pair and store it encrypted on the device. Next, it will use the `create_entity()` function from the WAVE module, which uses the WAVE CLI to create a new WAVE entity with the same password as chosen earlier. Finally, the entity hash of the newly created entity is stored in the config file along with the key alias.

This newly created user will be present in the WAVE system, but its entity hash and public key have not yet been registered in the BDMs. Client authentication will fail if the user tries to connect to the BDMs before the user has been registered. An administrator must do this registration. Administrators will be pre-registered in the BDM databases with write-permissions to a special URI for registering users. Figure 5.1 shows flowchart of what happens after the client is started.

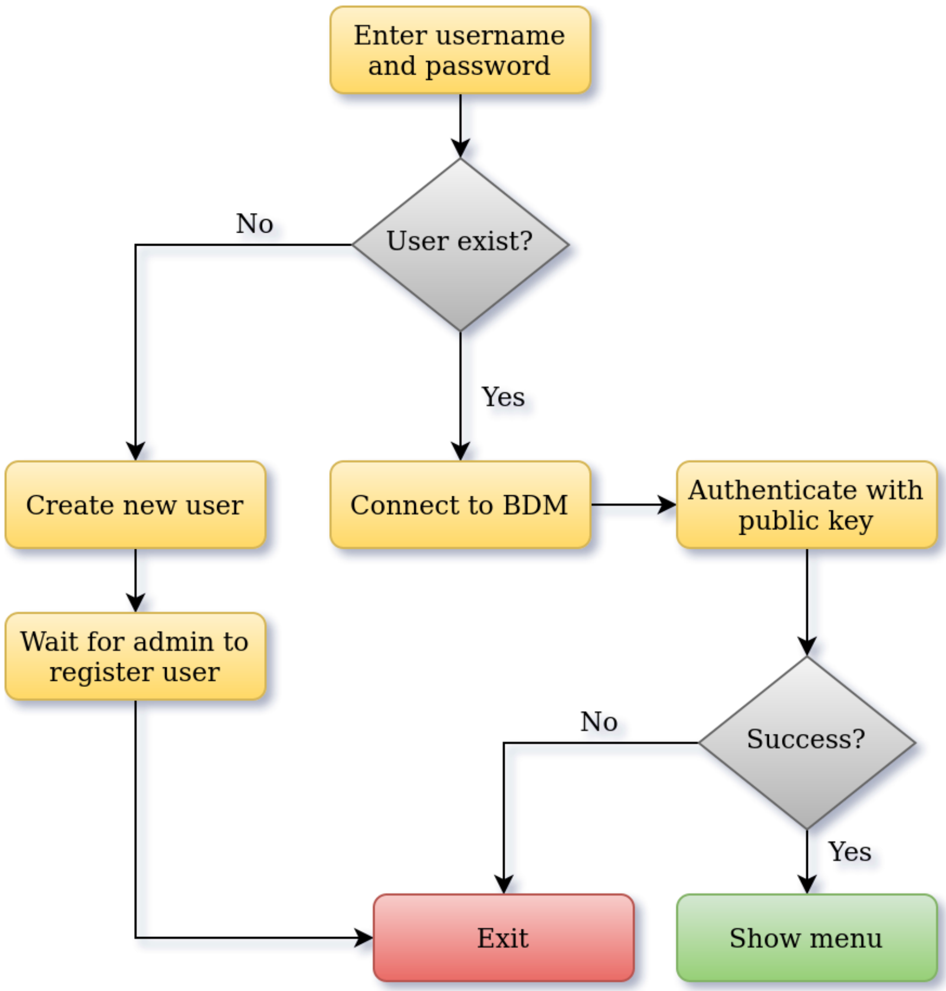

Enter username and password
User exist?
No
Yes
Create new user
Connect to BDM
Authenticate with public key
Wait for admin to register user
Success?
No
Yes
Exit
Show menu

Figure 5.1: Client startup flowchart

6 Evaluation

The purpose of this evaluation is to test and demonstrate that authentication and authorization works as intended. To do this, it helps to list some requirements.

- Users must register and authenticate with the blind data miners to get access.
- Users should have access to their own data.
- Users should not have access to other users' data unless access has explicitly been shared.
- Users should be able to share access to their data with other users.
- Users should be able to revoke the access they have shared with others.

Chapter Outline

- **Section 6.1** descibes the environment that has been set up to test the implementation.
- **Section 6.2** describes how registration and authentication was tested.

- **Section 6.3** describes the test and results for when users try to access their own data.
- **Sectiion 6.4** describes the test that demonstrates that a user can not request data from a URI that it is not authorized for.
- **Sectiion 6.5** describes the test that demonstrates access sharing.
- **Sectiion 6.6** describes the test that demonstrates access revocations.

6.1 Test Environment

The implementation has been tested using three BDM servers. All three BDMs are initialized with a WAVE entity that acts as an administrator to register new users. The idea is that this should be the initial state of the system once it has been set up at different organizations. All BDMs are running on the same machine. There are two reasons for this. First, it is more convenient, and it will not affect the functionality of the system. Second, there are no Unequivocable Log Derived Map(ULDM) storage present for reasons that are discussed in section 7.2 about limitations. Therefore, BDMs must share filesystem to have access to attestations and entities. The lack of ULDM means that auditing is not tested.

6.2 Registration and Authentication

First, we demonstrate that newly created users that has not been registered by an administrator yet will not be able to authenticate. We create a new user and immediately try to connect to the BDMs. The BDMs responds as expected with an error message saying they were unable to authenticate the user. This is because the users entity hash and public key does not exist in the BDMs' databases. The connection is then terminated.

Logging in with the administrator account, we will register this user. We have to manually input the entity hash and public key of the user with this client, but recall that the idea is that this will be read from a QR code. In this test they are copied and pasted it into the command line. The client generates a proof and sends a write request to the */users* URI with the new user. BDM validates the proof and stores the new user. Client gets a confirmation message that the request was successful.

Now when trying to connect to the BDMs with the new user, the authentication succeeded and the user is shown a simple menu for doing storage operations. The user is also shown the option to register users, but when trying to do so, the WAVE module will raise an exception when attempting to generate the proof because this user has not been given write-permissions to the */users* URI. In this case, the client will not send the request at all, thus saving the BDM from using resources to evaluate an invalid request. Even if a request is sent, the BDM will not receive a valid proof, and the request is rejected.

6.3 User access their own data

Users must have access to their own data. In this section, the system for authorizing users to access their own data is tested. The user that was created in the previous Section 6.2 is used.

This test is very simple. When the user was registered, the BDMs should have used the WAVE module to give full access to a unique URI like the one shown below containing all the data for this specific user.

```
"uri" : "<entity_hash>/*"
```

The client has the entity hash of the locally logged in user and this hash is automatically added to the beginning of the URI when the user tries to access data. A proof for this URI is generated by the client and sent with the request. All types of requests (read, write, delete and modify) were tested and all of them succeeded.

6.4 Accessing Unauthorized URIs

Users should not have access to other users' data. To test this, an additional user is created and registered. These users will be refered to as user #1 and user #2. We will try to make user #1 access user #2's data. Logging in with user #1, we enter the entity hash of user #2 we try to read data from. The client CLI has an option to manually enter the full URI without adding the hash of the logged in user to the beginning of the URI. Similarily to when trying to register users with a regular account, the WAVE module failed to create a proof and the client does not send the request.

Stealing Proof

We also try to simulate a situation where user #1 has been able to steal a proof that belongs to user #2, and use this to try to get access to user #2's data. Since the client does not store proofs on disk in this implementation, this had to be tested manually by creating the proof using the WAVE CLI which outputs the proof as a file that get stored locally. The code had to be modified to read this file and add it as a proof to the request. The request was sent to the URI for user #2. The BDM sees that this is a valid proof for the requested URI. However, the proof also contains the entity hash of the user that the proof belongs to. The authenticated user #1 that is sending this request is not in the proof, and therefore the BDMs respond with an error message saying that the proof belongs to a different user.

User #1 can not modify the proof because it is signed by the user #2's private key. The WAVE CLI will detect if the proof has been modified.

6.5 Access Sharing

Now user #2 will share access to his/her data and we run the same test as we did in section 6.4. This time the client is able to generate the proof and a request is sent. This time the request succeeeds.

6.6 Revocations

Users can also revoke shared permissions at any time. This is the final test in this evaluation. The access delegation from section 6.5 is revoked and once again we run the same test from section 6.4. As expected, the request fails.

7 Discussion

This chapter will discuss advantages and limitations of the design and implementation done in this thesis.

There will also be a section on future work, the performance of WAVE, and a conclusion on whether or not WAVE is a viable alternative to Keycloak.

Chapter Outline

- **Section 7.1** discusses advantages of WAVE and public key authentication.
- **Section 7.2** discusses the limitations of WAVE and design and implementation done in this thesis.
- **Section 7.3** discusses some future work for this thesis. Some are essential before the system can be used.
- **Section 7.4** discusses the performance of WAVE.
- **Section 7.5** discusses if the system designed in this thesis is something that should be considered by the chronic pain project, or if they should continue using Keycloak.

7.1 WAVE Advantages

This section will discuss the advantages of the design and implementation with WAVE done in this thesis.

7.1.1 Fully Distributed

One of the most considerable advantages of WAVE is that it is fully decentralized. There is even no need for peer-to-peer communication to generate and verify proofs. The only time there has to be communication between two entities is when sending a proof. The proving entity must send the proof to the verifying entity. Everything else happens through the WAVE service, which gets attestations and entities from the Unequivocable Log Derived Map(ULDM). At first glance, the ULDM might look like a centralized point of attack. The ULDM stores attestations and entity public keys and nothing that an attacker can use to grant themself permission to anyone's data. Attestations could, theoretically, be modified by an attacker to reference themselves instead. However, they are encrypted with public keys, so only those who have the private key can decrypt it. Even if an attacker could modify or delete an attestation, it would immediately be detectable since the ULDM Merkle trees would no longer be valid. Auditors have a copy of the latest map root head that is derived from previous versions. Suppose a storage provider modifies or removes anything from their ULDM and recalculates all the hashes. In that case, the map root will not be consistent with the auditors' version, and the storage provider can be permanently removed. As long as the majority of auditors are honest, the integrity of the system is ensured.

As discussed in chapter 4, the problem with Keycloak is that there is a single centralized point for attackers to exploit. There is no way for anyone to get hold of all users' credentials in WAVE because they are distributed across all users and stored encrypted on their devices. Even if an attacker can get access to a user's device, they would also need to crack the password used to encrypt the private key. Bruteforcing this encryption would take so much time that it is unreasonable to think that this is a real issue.

7.1.2 Public Key Authentication

The implementation uses public key authentication for both BDMs and clients. BDMs use the PKI(public key infrastructure) framework with X.509 certificates to authenticate, which is a system that has existed for decades and is by many considered the backbone of the internet.

Client authentication in this thesis does not rely on PKI. Instead, it is more similar to how public key SSH authentication is done. Approved public keys are stored on the BDMs, and clients prove that they have the corresponding private key by signing a random message generated by the BDM during the authentication procedure.

Public key authentication provides a cryptographic strength that extremely long passwords can not match. Traditional password-based authentication, like the one used by Keycloak, is prone to attacks like dictionary attacks where millions of common passwords are listed in a dictionary, and an attacker can quickly try all these passwords.

Usually, passwords are not stored in plaintext on the authentication server. Instead, the password is hashed, and the hash is what gets stored. If these hashes leak, an attacker can perform a rainbow table attack where pre-computed hashes and passwords appear in a table. These tables can have millions of passwords and their corresponding hashes, leading to passwords being cracked almost immediately. Passwords must be complicated enough to not appear in a dictionary or rainbow table to prevent this attack. These attacks are not a problem with public key authentication. There are no hashes involved, and it is practically impossible to derive a private key from a public key.

In addition, public key authentication frees the user from having to remember complicated passwords. One problem with the proposed solution using multiple Keycloak-servers is that users should ideally have different passwords for every Keycloak-server, which alone can be a reason for users not to use the system. I argue that due to the strength of public key encryption, the same key pair can be used for all BDMs to authenticate. However, it would not make it more difficult for the user to have different key pairs for every BDM since they do not have to remember them.

7.2 WAVE Limitations

7.2.1 WAVE in production

The implementation done in this thesis uses the latest version of WAVE (version 3)[1], which is not yet ready for production according to the developers. The documentation is lacking. Exactly how one sets up the ULDM storage and auditing is unclear. The WAVE CLI will output the attestations and entities as files that are stored locally, so it is only really usable in a system where

1. https://github.com/immesys/wave

processes share a filesystem or if entities or attestations are sent between devices. With thousands of users and permission delegations, it will quickly become unmanageable. If an entity that holds an attestation required by someone to form a proof is offline or decides not to send an attestation, the proving entity will not be able to form its proof. The ULDM storage is essential for this system to work, both because of availability and auditing that ensures the integrity of the WAVE graph.

Prior versions of WAVE that is in production uses an Ethereum blockchain for storage. This would not work here because it takes up to a minute to add an object to storage with Ethereum. Also, participating in a blockchain requires constant network bandwidth and CPU time, something a user should not be expected to provide.

7.2.2 WAVE is not for Authentication

WAVE is a framework for authorization and not a fully-fledged identity and access management system like Keycloak. Therefore, there had to be an independent authentication protocol on top of WAVE.

7.2.3 Loss of Private Key

The private key used for authentication is stored on the user's device. If the phone is stolen, lost, or if the user switches to a new device, the private key is gone, and the user will not be able to authenticate with the BDMs, and therefore loses access to all their data. Also, if the user loses its password for its private key or WAVE entity, there is no way to recover this password. Private key and WAVE entity will be gone forever, and access to data is lost. There has to be a system in place to recover users, or this system is too risky to put in production.

7.2.4 BDM as Namespace Authority

Having BDMs as the namespace authority can be an issue. It practically gives the organization that hosts the BDM the ability to grant permissions to anyone's data or revoke a patient's permission. If a patient's permission is revoked, then all downstream permissions will, in practice, also be revoked because it will not be possible to form a proof if the proving entity can not decrypt all attestations from itself to the namespace authority. Therefore, a solution where the patient is the namespace authority for its own data would be better.

7.2.5 Problems with Persistent Connection

The connection between client and BDM is always open as long as the client is running. This will probably not scale very well. With thousands of users online at the same time, there will be thousands of sleeping threads waiting for requests in the background. Sleeping threads does not consume any CPU time in most operating systems, but they do use memory. So some testing on whether this memory usage will be a problem should be done. Another thing to keep in mind is that there are only around 60 000 available port numbers in TCP, so a single BDM will not be able to have more than 60 000 clients online simultaneously even if it has infinite memory and processing power. 60 000 is a lot. It is unlikely that port numbers will be an issue in the chronic pain project, and it can easily be solved by having replica BDMs, but it is a limitation that must be considered.

7.3 Future Work

Before any further work on the system in this thesis is considered, it is crucial to figure out if WAVE version 3 will ever be ready for production or if the project has been abandoned. It has been over two years since the GitHub repository was updated. Perhaps it would be an idea to reach out to the researchers and developers to ask what their plan is. Assuming that a production-ready version will be released, a few things need more work before the implementation in this thesis can be put into production. This section will look at the most notable ones.

7.3.1 Revisit Authentication Protocol

The authentication protocol implemented is loosely based on SSH authentication. It has been implemented from scratch and may contain significant security holes that have not been considered. Before using it for the highly sensitive data that will be stored on BDMs, security experts should take a look at it. There are also probably some pre-developed solutions available that can be used instead. Perhaps the client authentication capabilities of SSL can be used.

7.3.2 BDM and Mobile Integration

The real BDMs have been implemented in Java. However, the implementation done in this thesis is written in Python, so if this is to be taken into production, the code must either be ported to Java, the BDMs ported to Python, or an API must be created to connect the two implementations.

Also, there have been no investigations on how to run the client on a mobile device. Kivy[2] is an open-source library for Python to develop cross-platform applications, but its capabilities have not been looked into. It is unclear whether the WAVE CLI and WAVE Service can be run on Android/iOS. WAVE is open-source and is written in Go[3], so further investigations on whether it is possible to compile it to run on Android/iOS must be done.

7.3.3 Private Key Backup

Public key authentication is, as discussed, a very secure authentication method. However, the loss of the private key will have significant consequences for the patient in the chronic pain project. He/she loses access to all data without the possibility to recover it. The BDMs' user database only stores an entity hash and public key, so there is nothing there to identify the owner of data if he/she does not have the private key.

Some way of securely backing up the private key should be implemented, or something that can tie the data to a patient other than the public key so that it is possible to create a new key pair for the patient if the private key is lost.

7.3.4 Proof caching

In this implementation, the client creates new proofs every time a request is sent. It takes about one second to create a proof, and the client must create one for every BDM it is connected to. If proofs can be cached by the client and reused, it would dramatically increase the client's performance. Technically it is not difficult to do this, but more investigations on whether it is safe must be done. As discussed in section 6.4, proofs are signed and can not be modified without being detected by the WAVE Service, so storing them should not be a problem. Also, the proof verification process detects if an attestation has been revoked after the proof was generated.

2. https://kivy.org/
3. https://golang.org/

7.3.5 Batch processing and Multi-threading

The BDMs are single-threaded in this implementation, which means that they can only handle one user at a time. To handle more users, the BDMs must be multi-threaded.

The client is also single-threaded. To increase client performance, the BDM communication could be handled by a separate thread running in the background. Also, to decrease the load on the BDMs, an idea is to process write-, modify- and write-requests in batches.

7.4 Benchmarking

There has been no formal benchmarking of the implementation because by simply observing the performance, WAVE performed much worse than what their paper claimed [4]. This can have something to do with how it has been set up and used here. As mentioned previously, the documentation is almost non-existent. It may still be interesting to discuss the results that the researchers of WAVE got when benchmarking.

In the real world, this would run on mobile phones and not on a laptop as was done in the evaluation. The original paper tested WAVE on devices that are on both ends of the performance spectrum. Table 7.1 shows some of the results.

Table 7.1: Object operation times (ms) [4]

Operation	**AMD64**	**ARMv8**
Create attestation	43.7	445
Create entity	8.9	88.5
Decrypt attestation as verifier	0.48	4.44
Decrypt attestation as subject	3.87	44.0
Decrypt delegated attestation	6.22	67.9

The interesting column here is ARMv8, which is benchmarks done on a Raspberry Pi 3. A Raspberry Pi 3 is not a very powerful platform, and a mobile phone can be expected to perform better. Also, if Apple ever put their M1-chip[4] in an iPhone, it will most likely even outperform the Intel i7-8650U that was used for the AMD64 benchmark since the M1 is much more powerful.

4. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

Figure 7.1 shows the time it took to build and verify proofs of different length.

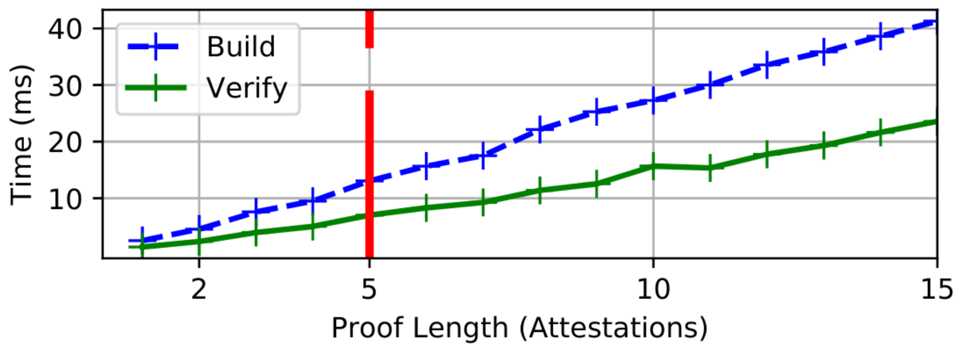


Figure 7.1: Proof build/verification time [4]

The vertical red line is the expected maximum proof length of a common application. It is reasonable to think that for most patients in the chronic pain project, it would not be much longer than 5. Therefore, around 14ms to build a proof and half of that to verify a proof should be fast enough to not cause any frustration by users even if the client can not cache proofs.

7.5 Keycloak or WAVE?

While the centralized design with Keycloak has some drawbacks, Keycloak is at least production-ready.

As it stands now, I would not go for WAVE. On paper WAVE looks very promising, but as long as the implementation is not production ready, and if it is unclear whether it will ever be, the best option is to continue using Keycloak for now. Keycloak is built around several proven industry standard technologies and protocols like OpenID Connect, OAuth 2.0, SSO and LDAP to mention a few.

Also, Keycloak servers can be hosted by anyone, and in the chronic pain project it would be hosted by the Norwegian Centre for E-health Research (NSE) behind their firewall. It is not a cloud-based solution like Firebase or AWS, so nobody outside NSE would have access to the data stored on the Keycloak server.

WAVE has a production ready version[5], but this uses a modified version of the Ethereum blockchain which is too slow (up to a minute to add something to storage). This version does not meet the requirement of a fast and responsive system as specified in section 1.1.

The proposed design with multiple Keycloak servers could possibly be used with a single password to authenticate with all servers if the client appends a random salt (which would be different for every Keycloak server) to the password when registering and also later every time the user logs in. A salt in this context is extra characters added to the password by the application before it is hashed and stored on the authentication server.

However, this raises the question of how to generate this salt. An idea could be to do something like the key-generation in identity-based encryption, where cryptographic keys are derived from a unique identifier. Salts could be generated similarly by having a known identifier as the seed to a salt generator. This way, the salts would available without having to back them up if they switch to another device.

5. https://github.com/immesys/bw2

/8 Conclusion

To conclude we repeat the goal from section **??**

> *The goal of this thesis is to look for a decentralized alternative for identity and access management that will be used with blind data miners in the chronic pain project. When a suitable alternative has been found, a prototype will be designed, implemented and evaluated according to the requirements specified in section 1.1*

WAVE was discovered when reading through literature on authentication and authorization. WAVE is a decentralized authorization framework with transitive delegation and it looked like a promising framework as a base for an IAM system in BDMs since it does not rely on a centralized server like Keycloak.

A prototype system was designed and successfully implemented that uses public key authentication for both BDM and client authentication. Authorization is handled by WAVE, which is also based on asymmetric encryption, to delegate, prove, and verify permissions. The system allows patients to share access to their data with other authenticated users. The evaluation of the system showed that it works as intended.

The conclusion, however, is that while WAVE is a very promising technology, it is not yet ready to be used in a production setting. The system implemented in this thesis was not able to get close to the performance that was stated in the WAVE researchers' paper [4], and therefore it does not meet the requirement of being a fast and responsive system. So for the time being, the chronic pain project should stick to using Keycloak until WAVE gets a proper release and documentation.

Bibliography

[1] Michel Abdalla et al. “Wildcarded Identity-Based Encryption.” In: *Journal of Cryptology* 24.1 (Jan. 2011), pp. 42–82. DOI: 10.1007/s00145-010-9060-3. URL: https://doi.org/10.1007/s00145-010-9060-3.

[2] *About Keycloak*. URL: https://www.keycloak.org/about (visited on 06/10/2021).

[3] Michael P Andersen. *WAVE Presentation*. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/andersen (visited on 04/03/2021).

[4] Michael P. Andersen et al. “WAVE: A Decentralized Authorization Framework with Transitive Delegation.” In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC’19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 1375–1392. ISBN: 9781939133069.

[5] Johan Gustav Bellika. *Decision support for personalized chronic pain care*. URL: https://ehealthresearch.no/en/projects/decision-support-for-personalized-chronic-pain-care (visited on 04/24/2021).

[6] A. Langley Ben Laurie and E. Kasper. *RFC 6962 - Certificate Transparency*. 2013. URL: https://tools.ietf.org/html/rfc6962 (visited on 04/03/2021).

[7] Arnar Birgisson et al. “Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud.” In: *Network and Distributed System Security Symposium*. 2014.

[8] Let’s Encrypt. *A nonprofit Certificate Authority providing TLS certificates*. URL: https://letsencrypt.org/ (visited on 05/02/2021).

[9] GDPR. URL: https://gdpr.eu/ (visited on 05/09/2021).

[10] Google. *How Log Proofs Work*. URL: https://sites.google.com/site/certificatetransparency/log-proofs-work (visited on 04/12/2021).

[11] Anne Håkansson. “Portal of Research Methods and Methodologies for Research Projects and Degree Projects.” In: *Proceedings of the International Conference on Frontiers in Education : Computer Science and Computer Engineering FECS’13*. QC 20131210. CSREA Press U.S.A, 2013, pp. 67–73. ISBN: 1-60132-243-7. URL: http://www.world-academy-of-science.org/worldcomp13/ws.

[12] Norwegian Directorate of Health. *Tryggere helseapper*. 2021.

[13] HIPAA. URL: https://www.hhs.gov/hipaa/index.html (visited on 05/09/2021).

[14] *How CT fits into the wider Web PKI ecosystem*. URL: https://certificate.transparency.dev/howctworks/ (visited on 04/03/2021).

[15] Sara Jelen. *What Are Certificate Transparency Logs?* 2018. URL: https://securitytrails.com/blog/what-are-certificate-transparency-logs (visited on 04/12/2021).

[16] Per Atle Bakkevoll Johan Gustav Bellika Elisa Salvi. *Data management plan. Decision support for personalized chronic pain care*. 2021.

[17] Hipaa Journal. *100% of Tested mHealth Apps Vulnerable to API Attacks*. URL: https://www.hipaajournal.com/100-of-tested-mhealth-apps-vulnerable-to-api-attacks/ (visited on 04/23/2021).

[18] Ben Laurie. "Certificate Transparency." In: *Commun. ACM* 57.10 (Sept. 2014), pp. 40–46. ISSN: 0001-0782. DOI: 10.1145/2659897. URL: https://doi.org/10.1145/2659897.

[19] James A. Martin and CSO John K. Waters. *What is IAM? Identity and access management explained*. 2018. URL: https://www.csoonline.com/article/2120384/what-is-iam-identity-and-access-management-explained.html (visited on 02/21/2021).

[20] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function." In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.

[21] Roberto Tamassia Michael Goodrich. *Introduction to Computer Security: Pearson New International Edition*. First Edition. Pearson, 2014. ISBN: 1-292-02540-9.

[22] Raluca Popa. *WAVE: A Decentralized Authorization Framework from Transparency Logs*. 2019. URL: https://simons.berkeley.edu/talks/wave-certificate-transparency-and-key-transparency (visited on 04/06/2021).

[23] Grand View Research. *mHealth Apps Market Size, Share Trends Analysis Report By Type (Fitness, Medical), By Region (North America, APAC, Europe, MEA, Latin America), And Segment Forecasts, 2021 - 2028*. 2021.

[24] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[25] Ronald Rivest and Butler Lampson. "SDSI – A Simple Distributed Security Infrastructure." In: *See the SDSI web page at http://theory.lcs.mit.edu/ cis/sdsi.html* (Aug. 1996).

[26] Adi Shamir. "How to Share a Secret." In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[27] Dawn M. Turner. *Digital Authentication - the basics*. 2016. URL: `https://www.cryptomathic.com/news-events/blog/digital-authentication-the-basics` (visited on 02/27/2021).

[28] UiO. *Tjenester for Sensitive Data*. URL: `https://www.uio.no/tjenester/it/forskning/sensitiv/` (visited on 06/02/2021).

[29] S. B. Wikina. "What caused the breach? An examination of use of information technology and health data breaches." In: *Perspect Health Inf Manag* 11 (2014), 1h.

[30] Wikipedia. *Merkle Tree*. URL: `https://en.wikipedia.org/wiki/Merkle_tree` (visited on 04/19/2021).

[31] K. Y. Yigzaw, A. Michalas, and J. G. Bellika. "Secure and Scalable Statistical Computation of Questionnaire Data in R." In: *IEEE Access* 4 (2016), pp. 4635–4645. DOI: `10.1109/ACCESS.2016.2599851`.

[32] Tatu Ylonen et al. *SPKI Certificate Theory*. RFC 2693. Sept. 1999. DOI: `10.17487/RFC2693`. URL: `https://rfc-editor.org/rfc/rfc2693.txt`.

[33] Kim Zetter. *DigiNotar Files for Bankruptcy in Wake of Devastating Hack*. 2011. URL: `https://www.wired.com/2011/09/diginotar-bankruptcy/` (visited on 04/06/2021).