Enhancing the security of an audit log

Kristoffer Smedlund
Abstract

Enhancing the security of an audit log

Kristoffer Smedlund

In today's society data breaches have become an all too common issue. Insecure databases allows intruders to access sensitive data. This thesis examines how to enhance the security of a relational database. The database is used as an audit log in a system platform at LeanOn AB. The security enhancements are made by implementing cryptographic hash sums to ensure that the data in the database can't be altered without detection, and by implementing symmetric cryptography to ensure that only authorized people can read and write the data. Some basic security features that the database management system provides are also used. The thesis project resulted in a database with an enhanced security compared to the old database design. This did however come at the cost of performance as the new design significantly lowered the scalability of the database.
Acknowledgements

I would like to thank LeanOn AB for giving me the opportunity to do my thesis at their company, working with their code. Special thanks to my supervisors Joakim Ek and Johan Ryhd for helping me with the project, and to everyone else at the company for making me feel welcome.
Contents

1 Introduction ........................................ 6
   1.1 Motivation and Goals ............................ 6
   1.2 Requirements .................................... 7
       1.2.1 Tamper proof ............................... 7
       1.2.2 Authorization ............................... 7
       1.2.3 General Data Protection Regulation ............ 7

2 Background ........................................... 8
   2.1 Symmetric Cryptography ......................... 8
       2.1.1 Advanced Encryption Standard ................ 8
   2.2 Cryptographic hash functions and checksums ........ 9
       2.2.1 Secure Hash Algorithm ....................... 10
   2.3 Relational Database Functionalities ............... 10
       2.3.1 Checks ..................................... 11
       2.3.2 Triggers .................................... 11
       2.3.3 Stored procedures ............................ 12
   2.4 Spring Framework ................................ 12

3 Design .............................................. 14
   3.1 Current design .................................. 14
   3.2 Design using secure techniques .................. 14
       3.2.1 Tamper proof ............................... 15
       3.2.2 Authorization ............................... 16
       3.2.3 GDPR ...................................... 16
       3.2.4 Schema changes ............................. 16

4 Implementation ...................................... 19
   4.1 Changes to the Server ........................... 19
   4.2 Changes to the Database ......................... 20

5 Evaluation .......................................... 21
   5.1 Security ........................................ 21
   5.2 Performance ..................................... 22

6 Related Work ....................................... 26

7 Conclusions and future work .......................... 27
1 Introduction

Since the early days of the Internet security has always been an issue [1]. As the Internet evolves and new technologies develop new ways of abusing them are developed. The threats that from the start were lone hackers have today grown into entire industries based on cybercrime [2]. This of course puts great pressure on every user of the web to make sure they are safe, and especial pressure on companies handling data.

Society has seen a big upswing in data breaches and hackings, as seen in figure 1. These increments are of course major threats to companies. For companies to ensure their data is safe from those threats many precautions have to be taken. One particular precaution, which this report will focus on, is to ensure that even in the event of a database intrusion, the data is still secure.

![Annual number of data breaches and exposed records in the United States from 2005 to 2017 (in millions)](image)

Figure 1: Annual number of data breaches and exposed records in the United States from 2005 to 2017 (in millions) [3]

1.1 Motivation and Goals

In LeanOn AB’s system platform “LISA” [1] there is an audit service which logs events that happens in the platform. Examples of such events are when a user is authenticated by the server, and when a transaction is processed. The purpose of this log is to be enable administrators to backtrack what has happened in the system.

The current storage for the audit events is a relational database, without any particular security features implemented at the database level which could stop intruders from modifying the data. This means that any client with access to the storage has the possibility to alter data without alerting the administrators.

This report aims to increase the security of this relational database based audit log. The goal is to create a database where altering of data is impossible without detection, and preferably not possible at all. Where reading data only is possible if you’re

---

1More information about the platform can be found at https://leanon.se/LISA
authorized, and where removal of data only can occur under secure conditions which confirms that the removal is executed by an authorized person.

For the purpose of making data only readable by authorized people symmetric key cryptography will be used. To guarantee that no changes has been made to the data checksums which calculate sums based on previous content will be used, as well as triggers. To ensure the validity of the database this thesis will make use of SQL rules and triggers.

1.2 Requirements

Most essential to this project is the requirement of a secure database. To concretize this requirement what is meant is the requirement to guarantee the authenticity of the data, as well as guaranteeing that meaningful interactions \(^2\) with the database is only made by authorized people.

1.2.1 Tamper proof

Tamper proof in this thesis means not changeable without being discovered. Since this database represents an audit log changes to the data, i.e. updates or deletions, are not desired at all. Ideally tamper proof should mean not changeable at all, but as explained momentarily other requirements complicate the matter.

By making the database tamper proof the data is guaranteed to be either authentic or obviously altered when reviewed by an administrator. This is true assuming that the data at entry is authentic, i.e. entered by an authorized person.

1.2.2 Authorization

The need for authorization is common for a majority of databases. Without proper authorization anybody with access can freely create and read data in the audit log. If an intruder were to gain access to the database ideally he would still not gain access to the plain text data.

1.2.3 General Data Protection Regulation

Since this database is essentially a log ideally it should be limited to an append-only style, where no updates and no deletions would be possible. There is however a new EU regulation named General Data Protection Regulation (GDPR) which applies to this database, which makes the requirements a bit more complicated.

GDPR is a regulation instated by the European Union. It’s main purpose is to standardize data protection laws for all EU countries. The aim is to give individuals more control over what companies can save about them[4]. The part of GDPR which is affecting the requirements of this database is article 17, which regulates what is called “the right to be forgotten”. Essentially this article states that a person has the right to erasure of their data if any of a number of conditions are met. \(^3\) For this report this means that an append-only database is not sufficient, erasure of data must be possible.

---

\(^2\) Reading encrypted data for instance would be an example of a non-meaningful interaction with the database, since it’s essentially equivalent to not reading at all.

\(^3\) For example if the person is no longer a customer of a company, the person can request that the company removes data tied to this person.
2 Background

In this section the techniques which are used to implement the requirements and secure the database are described and explained. First in section 2.1 there is an explanation of symmetric cryptography and the Advanced Encryption Standard algorithm. In the section 2.2 there is an explanation of cryptographic hash functions and the Secure Hash Algorithm. In section 2.3 some functionalities of the relational database model and the relational database management system are described. Last, in section 2.4 the Spring Framework for Java is described.

2.1 Symmetric Cryptography

Symmetric cryptography is an encryption technique widely used in practically all areas of computer science where encryption is necessary [5]. This encryption technique is very fast at both encrypting and decrypting data compared to other techniques, and is therefore commonly used to encrypt files and data. The technique can be used anywhere where both the sender and the receiver of the encrypted message has access to the same secret key. As an example, to ensure privacy between two parties in chat applications such as Whatsapp, symmetric cryptography is used[6].

Symmetric cryptography builds on the concept of using bijective encryption where the same key is used to both encrypt and decrypt messages. The encryption can be defined as follows assuming an encryption algorithm E, a decryption algorithm D, a message m and a key k:

\[ E(m, k) \rightarrow c \]
\[ D(c, k) \rightarrow m \]

Where c is a cipher text.

The encryption and decryption algorithms can be publicly known without losing any security since the encryption relies on the secrecy of the key. Similar to cryptographic hashing functions which are described in the next section, symmetric encryptions hold the property that when given a cipher text c (or in the case of hash functions a checksum c), it is infeasible to calculate the message m (assuming one does not have the secret key).

2.1.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a widely adopted symmetric cryptography standard. The algorithm on which the standard is based was chosen in a competition hosted by the National Institute of Standards and Technology (NIST) started in 1997 [5]. This competition went on for multiple years with several elimination rounds where experts analyzed all the contributions and decided which are good enough to proceed to the next round. Finally in the year 2000 NIST settled on an algorithm named Rijndael developed by J. Daemen and V. Rijmen [5].

The algorithm is considered to be very secure as well as fast, making it a very suitable standard. Along with most encryption algorithms it follows the design principle of open design as described by Saltzer and Schroeder’s design principles; the implementation is common knowledge and there are no dependencies on keeping the design a secret. Instead the security relies on the key being secret [7].

---

4The biggest chat application in 2018
Rijndael is an iterated block cipher. This means that the message to encrypt is divided into blocks, in Rijndael’s case blocks of 128 bit length (where if the message is not a multiple of 128 it is padded) and then each block is iteratively processed in a certain number of rounds depending on the length of the key used. Key lengths in Rijndael are 128, 192, and 256 bits (usually denoted as AES-x, for example AES-128 if a 128 bit long key is used) and the number of rounds for respective key length are 10, 12 and 14. In each round certain steps are executed on the bytes of the block. Since there are 128 bits in a block, there are 16 bytes which during the rounds are treated as 4x4 matrices. Before the first round the private key is expanded to derive individual keys for each round. Thereafter four steps are executed each round:

1. Every byte in the matrix is substituted using a substitution box (applying a function which maps the current byte to a substitute byte).
2. Next each row in the matrix is shifted around in a predetermined order.
3. There after each column is multiplied with a polynomial to produce a changed column.
4. Finally each byte is combined with the derived key of that specific round by using the XOR operation. This is the only operation which involves the key.

All these operations have their respective inverse operation, which is used for decryption of the cipher text.

There are currently no known efficient ways of cracking AES, thus brute forcing guessing the key is the only way. Using a 128 bit key the number of possible keys is $2^{128} = 3.4 \cdot 10^{38}$. Assuming one has all the hashrate power of the bitcoin mining network (which are used for SHA-256 hashes, but assume it can be used just as well for AES decryption) one could make $2.8 \cdot 10^{19}$ guesses a second [8]. Assuming this enormous computing power, the key would on average still take longer to guess than the estimated age of the universe.

### 2.2 Cryptographic hash functions and checksums

A hash function in general is a function which maps some input of any size to an output of a fixed size. There are many use cases for hash functions, for example the well known data structure hash tables, making database querying more efficient by mapping search parameters to places in memory (buckets) for direct lookup, and password storing [9].

A hash function can be defined as a function $F$ which takes some input $M$ of arbitrary length and returns a (usually compressed) hash value $V$. This can be formulated as:

$$F : M \rightarrow V$$

There are many forms of hash functions, this report will focus on cryptographic hash functions since they possess the desired properties for this use case. The calculated hash value will be referred to as the checksum.

There are some desired properties of a cryptographic hash function which this report will take special use of:

---

5The peer to peer network used by the cryptocurrency bitcoin to confirm transactions by computing hashes. Because of all the contributing computers the network possess an enormous computing power.
For any given message $M$, it is extremely likely that the calculated checksum $V$ is unique. A slight change to $M$ makes the new checksum $V'$ completely uncorrelated to $V$.

$$\forall m_1, m_2; m_1 \neq m_2 \rightarrow V_1 \neq V_2$$

There is however a small risk that $V_1 = V_2$ since the amount of possible messages are normally greater than the set of possible checksums. This is known as a collision. The risk is however negligible for cryptographic hash functions.

- For any two given messages $M_1, M_2$, if they are equal then their checksums will be equal. This is always true for cryptographic hash functions.

$$\forall m_1, m_2; m_1 = m_2 \rightarrow V_1 = V_2$$

- Hashing is a one way function. Given a checksum $V$, calculating the hashed message $M$ from $V$ is infeasible without brute force hashing all possible values of $M$ to find a $V'$ such that $V' = V$.

Since the hash function will be used extensively in the implementation, it is also desired that the hash function is very fast at computing the checksum, a property which is very common among cryptographic hash functions.

Checksums are very powerful and used extensively to guarantee the integrity of data. A previously calculated checksum can be stored and when data integrity verification is required the checksum can be recalculated and compared to the previous checksum, if they are the same then the data integrity is verified. Their main purpose in the report is to act as a guarantee that no data in the database can be changed without discovery.

### 2.2.1 Secure Hash Algorithm

Secure Hash algorithm (SHA) is a family of cryptographic hash functions developed by NIST. [9] This report will make use of the SHA-256 algorithm from the SHA-2 set. The numbers in the name SHA-256 represents the digest length, i.e. the length of the output of the hash function.

The algorithm operates on blocks similar to AES. The block size is 512 bits and messages which are not a multiple of 512 are padded. The algorithm can be described to work in two stages, preprocessing and hash computation. The preprocessing stage handles the padding, initialization of values used in the hash computation, and parsing the message into blocks. The hash computation stage iteratively generates a series of hash values by using functions, constants and word operations. [10]

Note that whether there even exists true one way functions is currently unknown, but currently there is no known way of efficiently computing $x$ given $y$ where $y = H(x)$.

### 2.3 Relational Database Functionalities

Relational databases are the traditional databases dating back to the 1970’s [11]. They are used to store structured data, i.e. data where the type and structure of data is known. Relational databases stand out from other types of databases partly because of the powerful Relation Database Management Systems (RDBMS) which provides lots of functionalities to ease the use of a database and enhance its function. Another contributing factor to why relational databases are so popular is their use of the Structured Query Language (SQL). SQL is used to communicate with the database.
database, for instance when querying the database and when defining schemas.

This report will make great use of some functionalities which the RDBMS provides, described below.

### 2.3.1 Checks

The RDBMS provides a tool for asserting that the data in the database hold certain constraints at all times, called checks. Checks allow a user to set constraints on individual columns. When a create or update query is executed the check is evaluated into a boolean value. Should the check fail the RDBMS will deny the query from updating the state of the database. Do however note that not all RDBMSs support checks, but the same functionality can be achieved (although in a somewhat clumsier way) by using a soon described Trigger. [12]

Below is an example of a table creation with a constraint on the price attribute to always be greater than 0.

```
CREATE TABLE fruits (  
    fruits_id integer,  
    fruit_name text,  
    price numeric CHECK (price > 0)  
);
```

When using checks one should be wary of the extra computational overhead that might be added if the constraints are complex, which might slow the database down.

### 2.3.2 Triggers

An active database is a database which is built with the Event-driven architecture software design pattern in mind, often especially using the Event Condition Action (ECA) structure.

To create these rules in relational databases triggers are used. Triggers is a way for the database to react to events in the database. The ECA model explains how triggers work in general.

- **Event** - What event happened that made the database invoke the trigger.
- **Condition** - The condition that evaluates the event and determines the action to take.
- **Action** - What SQL statement, transaction, or even external program should be executed as a result of the condition.

There are three different types of triggers which differentiate by when they are executed: The before trigger execute before the event is processed and is typically used to check constraints on the attributes. The after trigger is executed after the event has been processed and can be used for example as a tool for auditing. The instead of trigger executes instead of the event, and can for example be used to update other tables instead of the queried table, or to deny certain operations.

An example of a trigger on the previously declared table can be formulated as follows:

```
CREATE TRIGGER check_fruit_price  
BEFORE INSERT OR UPDATE ON fruits
```
FOR EACH ROW
BEGIN
IF NEW.price < 0
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Price cannot be a negative number'
    END IF;
END

This trigger checks the price constraint. If it’s negative the trigger will signal the SQLSTATE 45000 error, "Unhandled user-defined exception condition" with the specified custom message.

2.3.3 Stored procedures

A stored procedure is a subroutine accessible by the RDMBS as well as by any application with access to the RDBMS. What differentiates a stored procedure from an application function is that a stored procedure is stored and executed by the RDBMS on the server side, while the application function is executed on the client side by the application.

Stored procedures can be very useful in certain situations. When a database program is needed by several applications, storing the procedure on the server reduces code duplication and improves modularity. In some cases large amounts of data from the database might need to be processed, but only some small value needs to be returned. In this case processing the data on the server side reduces the amount of data that needs to be transferred. Stored procedures can also be used for security. By restraining the permissions of the application to only allow querying of views and to executing stored procedures the amount of harm an application can cause by querying the database with flawed SQL statements can be significantly lowered.

Stored procedures can differ a lot in complexity, they can be as simple as SQL retrievals and updates, or complex functions written in general-purpose programming languages (assuming the specific RDBMS supports such functions).

2.4 Spring Framework

The Spring Framework is the leading enterprise Java development framework [13]. It was initially released in 2002 as a reaction to Java 2 Enterprise Edition (J2EE) which many developers felt was much too complicated. At the time the Framework’s focus was mainly to provide what is today known as the Spring Core and especially inversion of control, explained in detail below. Today the Spring Framework provides lots of utility ranging from data access and integration components, to model view controller components, to testing components.

Central to the Spring Framework is the idea of inversion of control. Inversion of control is a technique that is used to externalize the creation and management of component dependencies in an application. This decouples objects from each other leading to simpler code, which for example reduces the need to change multiple classes solely because one class was changed. Decoupling the objects also makes the code easier to test.

---

6Signal is the way to return errors in SQL.
In a Java application the usual way of creating an object is by using the new key-word. Assume Example is an interface and ExampleOne is a class implementing the interface. An instantiation of an ExampleOne object can be done as following.

```java
public class Test {
    Example ex;

    public Test() {
        this.ex = new ExampleOne();
    }

    public void doWork() {
        ex.doSomething();
    }
}
```

In big applications this technique can be an issue. The Test class has to look up its dependencies. Suppose one would want to switch dependency to ExampleTwo in the Test class. This would require a recompilation of the application. This would be a lot simpler if the Test class would not have to deal with dependencies but instead simply have them injected by the spring container. The following is an example of the same class using Spring.

```java
public class Test {
    @Autowired
    Example ex;

    public void doWork() {
        ex.doSomething();
    }
}
```

Here Test receives the dependency from an external source, and does not have to do anything except declaring that it is to have its dependency injected by using Java annotations. In this simple example nothing more needs to be done, but in larger applications there might be need for a configuration file in form of an XML document or a configuration class to specify how the container should inject dependencies.

With all the power that the Spring Framework brings, it also has some drawbacks. A very common complaint has been that Spring requires too much configuration. The Spring Boot project aims to reduce the need for configuration by providing the basic configuration themselves while still allowing a user to override the provided defaults if necessary. Spring boot also provides features which are common to many applications, such as security, metrics and auditing.
3 Design

In this section the design of the current database will be explained, as well as the design for the secured database including some discussion regarding the design decisions.

3.1 Current design

When certain kinds of events occur in the system (for example logins, scheduled tasks are executed, accounts are created) these are logged. The logged data about the event is written as JavaScript Object Notation (JSON) \(^7\). The data contains a set of properties in the format "name: value", where each property contains some useful information about the event. Each event can have any number of "name: value" fields.

Along with the data some metadata about the event are stored. This includes its unique id, the date when the event was created, the type of the event and the username of the user that committed the action which caused the event (called principal).

The current database has two tables. AUDIT_EVENTS holds all events that has happened in the system. For each event the metadata is saved in AUDIT_EVENTS, while each "name:value" field is stored as an individual row in the AUDIT_EVENT_DATA table. Note that the AUDIT_EVENT_DATA is a weak entity, all rows in the table are connected to a row in AUDIT_EVENTS. There can only exist a row in AUDIT_EVENT_DATA if there exists a row in AUDIT_EVENTS with the same event_id. Many rows in AUDIT_EVENT_DATA can be connected to one AUDIT_EVENT, a so called many to one relationship.

![Current database schema depicted as an ER-diagram.](image)

The data is stored as plain text. With access to the database anyone can freely create, read, update and delete rows unnoticed. Since this database is used as an audit log, which purpose is to log events in the system so that an administrator can trace events both updates and deletions of rows are major risks.

3.2 Design using secure techniques

This section discusses the design of the altered database where the data is stored in a secure way to comply with the stated requirements.

---

\(^7\)A very common format for saving data used by large parts of the industry as a standard, with support from most large programming languages.
3.2.1 Tamper proof

One requirement of the secured database is that the database should be tamper proof. Ideally this means that there should be no way of altering what is already in the database, essentially making it an append only database.

To achieve this two techniques will be used. SQL triggers and checksums.

SQL Triggers can as previously described be used to create an active database which reacts to events. Triggers can be set on certain types of queries, in this case on updates and deletions. To guarantee a tamper proof database the triggers on any update or deletion should rollback the query. Using this technique no updates or deletions can be made on the database while the trigger is active.

If however, a trigger for some reason does not prevent an update or deletion, there is no way of knowing what or if anything has been changed. In this instance the use of checksums is highly applicable. Checksums guarantee that no changes to the database can occur without being discovered. They can however not restore data if changed, but instead informs an administrator of what data is not authentic.

By calculating the checksum of a row we know what data the row holds. By including a checksum field in each row of a table we can guarantee the authenticity of that row. There is however a flaw with this approach. If the checksum of a row is stored in the same row as the data, one could simply change the data and then recalculate the checksum. This makes it a bit too easy to make updates to the database without
discovery, and minimizes the impact checksums have.

If checksums instead are calculated on the previous row, a chain of checksums is created. As depicted in figure 3, the checksum field of a row can be calculated on the entire previous row. A simple change to a row now requires an attacker to recalculate the checksum stored in the following row. But that means changing that row, which in turn requires the checksum in the following row to be recalculated. This is propagated all the way up to the most recent entry in the database. Thus, a simple change to a row turns in to the exhausting task of recalculating each following checksum of the database. By sometimes storing the latest checksum in an inaccessible place, an administrator could verify that the checksums have not been changed.

3.2.2 Authorization

The requirement of authorization protects the database from unauthorized usage. As previously stated, updates and deletions on rows in the new database is not possible. But there is still the issue with unauthorized people being able to write and read to the database.

By encrypting sensitive data one can assure that only authorized people (people with access to the encryption key) can read the data. This encryption also controls writes to the database. If one were to write to the database with the intention of creating false data, one would still have to encrypt it in order to comply with the database design. But when encrypting, the key used still needs to be stored along with the other keys for it to be authentic.

The key cannot be stored in the database, since if the database is compromised it is probable that the table which would store the keys is compromised. Instead, the keys should be stored at a separate location. There are lots of software available which solves this problem, called Cryptographic Key Management Systems [14].

3.2.3 GDPR

As previously discussed GDPR requires that deletions in the database have to be possible for compliance. The data that has to be removed is the sensitive data which can be linked to a person. As described in the previous section encryption is used for authentication reasons. The use of this technique does however present an opportunity for deletion of data without compromising the append-only database design decisions previously described.

By deleting the stored symmetric key in such a way that there is no way to recover it, the encrypted data becomes unreadable, essentially deleted (see the discussion of cracking AES, in theory the data is decryptable by brute force guessing the key, but in practice this is extremely improbable). This approach is especially useful for GDPR compliance, since it not only deletes the requested data from the running database, but also from any backup which might store the same data (which is a requirement from GDPR).

3.2.4 Schema changes

To comply with the previously described changes, the database schema has to be changed.

When designing the secured databases there was a decision between two alternatives of design.
The first alternative was to have only one table where all the properties are kept in a
single table, as an opposite to the old design which had one table for metadata and
one table with entries for each data property. In this approach all data is stored in a
single field. The advantage of this approach is it’s simplicity. Since the data needs to
be encrypted, having all data in one field and only encrypt that field is simpler than
having to encrypt each data property stored in the AUDIT_EVENT_DATA table.
This approach also makes the process of calculating checksums for data authenticity
a lot simpler since it only requires hashing one row, as opposed to calculating a check-
sum over the metadata row and all rows with data connected to the same event.
The drawback of this approach is that the database schema no longer supports search-
ing on data property names. For example, previously a property could be defined in
the database table AUDIT_EVENT_DATA as:

```
event_id: 13
name: status
value: SUCCESS
```

With this design one could for example query the database to get all events with the
property "status", to see if they were successful.

The other design alternative was to keep the current schema design of two tables
where the data has its own separate table. This has the advantage of keeping the
ability to query on properties. This however comes at some costs. Compared to the
previous alternative, this approach would be more complicated since each row would
need to be encrypted individually. This approach would also complicate the check-
sum calculation because of the multiple tables instead of one single table. Since the
principal field needs to be encrypted as well, this approach requires that the principal
field is broken out into its own table formulated as

```
event_id
principal
```

Which further complicates the database schema design.

When considering these alternatives it was found that the complication of the second
alternative was not worth the advantage gained by being able to query on properties.
It was concluded that the initial design of being able to query on properties was a
design choice which in practice was never used. Therefore, the first alternative was the
obvious choice.

Figure 4 depicts the schema of the first alternative, containing only a single table. The
primary key event_id and the fields date and type are kept from the old design. The
DATA field is new and contains all the properties related to an event. The data field
is encrypted using AES. The field also contains the principal field from the previous
design, since this information needs to be encrypted and there is no need for it to be
kept separately. Lastly, the table has a checksum field containing the checksum of the
previous row as discussed previously.
Figure 4: ER Diagram of the new database design
4 Implementation

The server is written in Java using the Spring Framework [13]. It is written using a micro service architecture meaning that it is a stand alone service running entirely on its own with the sole purpose of managing audit logging. For clients to communicate with the database the server has a Representational State Transfer (REST) [15] API implemented through which clients can send Hyper Text Transfer Protocol (HTTP) requests to put or get data. REST is a standard for writing API’s and is very well established.

The data which is sent from and to the clients is the audit events. These events are represented using the Spring Boot AuditEvent class which holds the information:

```java
AuditEvent(
    Instant timestamp
    String principal
    String type
    Map<String, Object> data)
```

These fields correspond to the columns of the database described previously. The data which when stored in the database using the old design is represented as it’s own table (as shown in figure 2) is in Java represented using the Map data structure, corresponding to the ”name: value” representation of both JSON and the audit_event_data table.

In the server the received events are parsed into a special object with the purpose of mapping the object-oriented domain model to the relational database model. Without this mapping problems derived from object-relational impedance mismatching (i.e. problems due to the differences of the models) might appear such as data type differences and structural differences. For example, in the object-oriented domain model it is common for objects to consist of other object, such as the Map object in the AuditEvent object just described. To solve this the Map object is mapped into it’s own table in the relational database model.

4.1 Changes to the Server

Since the server is already a finished project there was a lot of code to read and understand before being able to change anything. Using the Spring framework comes with many great advantages, but for a person with limited Spring experience the code can be difficult to understand. Once accustom to the Spring framework, reading and changing the code was quite straight forward.

The main changes to the server was when parsing from the AuditEvent class form to the persistent form since all the presented changes only concern the database and how the data is stored there. Preferably the code for the rest of the server should remain unchanged as much as possible.

Two big changes were implemented in the parsing. First, the principal field and the data field is now encrypted using AES with the AES implementation from the javax.crypto standard library. The implementation of encryption introduced a higher level of complexity to the writing and retrieval of data from the database. When writing data, a key must be generated, data must be encrypted and the key must be stored securely. When retrieving data, the key must be retrieved securely, and the data needs to be decrypted. On several occasions
the server retrieves a list of principals from the audit_events table and uses that list for comparisons for authorization reasons. This was previously a relatively cheap solution in terms of processing required, but after the change such retrievals requires decrypting each queried row which instead is a very expensive solution.

The second big change is the checksum calculations on previous rows. This is implemented by retrieving the latest written row from the database, i.e the previous row, and calculating a SHA-256 checksum on it. Unlike the encryption change, this change does not impose any large changes to the rest of the server since it does not interfere with anything previously implemented.

4.2 Changes to the Database

Since the server contains the logic which encrypts the data and performs the checksum calculations, the only major database related change which needed to be done was to update the schema to adhere to the changes made in the server. The most difficult database related task was to setup the initial database and get it to work with the existing project. Thereafter changes to the database was easy to implement and test. The DBMS used in this project is PostgreSQL, which is a very common DBMS.

As described in the design section, the schema has been changed as shown in figure 4. Implementation-wise this means nothing more than changing the SQL schema and the corresponding Java object which is used by the server to map data to the database.

Implementing the database triggers that deny updates and deletions was trickier than anticipated. Although easy in theory (and seemingly easy when looking at the amount of code) finding the right statement for the PostgreSQL DBMS and getting it to work was not trivial. The final solution was to define an SQL function which returns a trigger with the sole purpose of terminating any query. This trigger function is then called by a before update or delete trigger for each queried row.

As mentioned previously the ideal storage for keys is a Cryptographic Key Management System (CKMS), which not only stores the keys but also securely generates and removes keys. In this implementation, however, because of time and money constraints instead of a CKMS the keys are generated in the server and stored in the database. Since this is only a proof of concept this is fine, swapping this implementation to an implementation using the CKMS is very simple since the code is already written, the only change are swapping the databases and letting the CKMS generate the keys.
5 Evaluation

In this section the overall design and the implementation will be discussed and evaluated.

5.1 Security

Compared to the old design the database is much more secure. Previously all types of querying the database could be done without needing to be authenticated or authorized. With the new design insertions are possible but there’s no point in inserting unless one has access to the key storage as well. Updating and deletions are denied by triggers, and even if these triggers were to be circumvented the implemented checksums would still detect any changes. Finally, reads are possible but only meaningful if one has the key to decrypt the data.

Although the database’s security has been significantly improved, there are some security issues with the new solution as well.

A vulnerability in every system based on encryption is how the encryption keys are stored. If an attacker gains access to the key database that person would be able to read and write to database without detection. However, the attacker would still not be able to update already written data. This report’s main concern is keeping the database tamper proof, so for this project this is fine.

Calculating checksums of data is a very good way of guaranteeing authenticity. The chain-like technique used in this report promises that the previous row in authentic. The checksum technique still has issues. The checksum can be recalculated for the entire database without a trace if one is not keeping track of latest checksum. A decent way of solving this issue is by always storing the checksum of the latest row in memory. Since the checksum needs to be calculated there is no redundancy. The issue with this solution however is that it assumes that the row has not been changed. Preferably when a new row is to be entered into the database the checksum of the latest row should once again be calculated and compared to the stored checksum. This does however introduce yet more overhead to the application. By writing the memory stored checksum to the database without comparisons, there is at least a guarantee that when re-calculating all the checksums to check the database’s integrity, an alteration of any row would be discovered.
5.2 Performance

A trade off between performance and security is always present when designing a system. As a rule of thumb, the more secure, the slower the application will be. This project is no different. While implementing the design changes a lot of performance related concerns appeared. These concerns are discussed in this section.

All tests in this section are done using a MacBook Pro 2017 with an Intel Core i7 7:th gen. 3.1 GHz quad core processor and 16 GB RAM.

Figure 5: Line graph comparing the time it takes to push different amounts of data rows to the database using the old design.

With the original design load testing showed that on average the application could push 2330 database entries per second. Do note that this is single threaded, it is also possible for the original design to run in parallel where only the DBMS limit how many entries a second can be pushed.
On average the new design managed to push 1700 entries to the database per second as shown in figure 6 below. Since the old design averaged 2330 pushes a second the performance decrease is about 27%.

However, the old design can run in parallel without any issues, but the new design is limited to one thread only. This is because of how the checksum security feature works. In a database row, the checksum stored is the checksum of the previous row. Since the checksums form a chain where each checksum has to be over the previous row, there must be a known, synchronous, order for the rows to be added in. Using several threads, there is no way to know which thread gets to execute before the others, and therefore no way of knowing the order of data pushes. Using multiple threads with this implementation would mean that some rows have their checksum stored in multiple other rows (which all saw that row as the latest row), while other rows checksums are not stored at all.

Being able to run an application in parallel is crucial for scaling. Limiting the application to one thread means that there is a ceiling which once reached would cause delays in the system and even drops of data.

One solution to this problem, which improves the scaling but does not deal with the core problem, is to run multiple instances of the application, essentially letting each application act as a thread. If for example four applications are running, events are shared so that every application receives 1/4 of the total amount of events. The problem with this solution however is that searching the database would not be the simple task of querying one database but instead four, lowering searching performance.

The choice of cryptographic hash function for this project was based on the currently most popular hash function, SHA-256. It is a very secure algorithm, unlike similar cryptographic hash functions no collisions have been found. However, when analyzing what the purpose of checksums are for this report, it is clear that although security is important, i.e. there must be few collisions, it is not a big security risk if some known
collisions do exist.

The scenario where the known collisions harm the integrity of the database is when a row’s checksum collides with a checksum of another message. In theory, this would mean that a row could be replaced by the other message without discovery. However in practice the risk that the other message would conform to the database schema design is low, but in theory this is still a concern.

Then the question of performance versus security arises, how much performance can be gained by switching to a faster cryptographic hash function such as MD5? Below is a figure comparing pushing data to the database while using SHA-256 and while using MD5.

![Comparison between MD5 and SHA-256](image)

Figure 7: Diagram comparing the performance of MD5 and SHA-256 when pushing 25000 entries to the database.

As seen, SHA-256 is slower by 0.749 seconds, or about 5.3%. Although not massive, this means an average of 90 more entries a second.

Another performance concern is that when writing an entry to the database, for the checksum calculation a read of the previous row has to be executed. This design decision means that for every write to the database, there is automatically also a read. This could put a lot of stress on the database and the question is if it is really necessary. As discussed in the previous security section the checksum could be stored in memory, but that would lower the security until a validation of the database is done.

The checksums are great for ensuring the authenticity of data. There is however a concern regarding how to validate them. In theory it is easy, just re-calculate the checksums and ensure that they do not differentiate. In practice this process can be very computationally heavy, and might take a very long time. The question then arises of how often the checksums should be validated, and if all of them (i.e from the absolute first entry) should be validated or if at some point the events can be moved to a different database as final storage and assumed to be correct but not tested in the original database.

Another solution would be to validate each update to the database to make sure that
the update only appends data. This would lower the performance of database writes, but would remove the need running the validation process since data is guaranteed valid.
6 Related Work

Since databases are such an important part of practically all applications, a lot of research has been done to enhance them. It’s often crucial to keep the data secure, so naturally there has been tons of research done to improve the security part of the technology, both by refining old techniques and by inventing new ones.

This project is inspired by the blockchain technique which has become a huge talking point in the last couple of years. The blockchain developers take great pride in the blockchain security, which guarantees authenticity of data with its security by design. A large part of the security implemented in this project is the checksum chain, which is very similar to that of how a blockchain guarantees authenticity.

There exists some variations in different implementations of the blockchain. This report will describe the blockchain technology as implemented by the popular cryptocurrency "Bitcoin", which was first to use the technology and is currently the biggest application based on it.

A blockchain is a list of blocks linked together [16]. Linking is done by calculating and storing a checksum of the previous block, thus creating a chain of blocks which can validate the authenticity of data from the last to the first block.

This report uses a centralized database to store the data. Blockchain instead (in most implementations) use a distributed database where each client in the network has its own copy of the database. This way each client has a history of the database and all the blocks, so when a broadcast with an update of the blockchain is sent each client can easily validate if the change is legit or not. Thus by using a distributed database the responsibility of guaranteeing the authenticity of data in the blockchain is not put on one centralized client but instead each client has the same responsibility.

New blocks are added by collecting data which is being distributed in the network and write it into a block, then distribute that block to the network. Each client adds the block to the chain assuming the block is valid. Sometimes two blocks can be sent at the same time, creating different versions of the chain for different clients in the network depending on which distributed version the client got first. This is solved when the next block is broadcast, when that chain will become the standard for everyone.

To avoid that dishonest clients pushes blocks freely there is some proof of work that needs to be done to ensure that one single client can’t create a lot of blocks in a row. In the case of the bitcoin blockchain, this proof of work is based on trying to compute a problem, where the first client to compute the problem gets to distribute its created block.

When joining a network using the blockchain technology one can check the integrity of the network by calculating the checksums of all the blocks. If there are many blocks this process can take a lot of time. The same problem is discussed in the evaluation section of this report, since the checksums of the database need to be validated at some point. Note however that in a blockchain one can not change previous blocks without also changing all blocks after, up until the current block. Because of the proof of work security measure this is highly unlikely, and would require the attacker to have an enormous computing power. In this project however, an attacker could change a row (assuming triggers are off) and would not need to recompute the whole chain. The change would be discovered upon recalculating the checksums, but until then unnoticed.
7 Conclusions and future work

This project has been both challenging and interesting. I got the opportunity to work with an existing code base using software techniques that were entirely new to me. Security is a very interesting subject, and I’m glad that I got the chance to deeply dive into the subject. This report shows both what’s to gain from cryptography by making the database more secure, but also what negative impact it has in terms of application complexity and performance.

The task was to make the relational database more secure. This was tackled by implementing techniques to guarantee both authenticity and authorization. These techniques were cryptographic techniques, namely cryptographic hash functions to guarantee authenticity and symmetric cryptography for authorization. This report also made use of the DBMS provided trigger functions to create an append only style database, acting as a first line of defense against anyone who tries to update or delete from the database.

The problem of performance versus security arose since the introduced changes lowers the performance of the server a lot, which means that scaling of the database is severely impacted.

The work done in this report has improved the security significantly, and while not perfected its an interesting initial research report regarding relational databases which are secure by design, rather than secure because of how the application interacts with it.

The biggest flaw in the solution presented in this report is the performance. Limiting the database writes to a single thread means that the only way to scale the application is by processing power, which is not infinite. This flaw appeared because when writing to the database the server needs to retrieve the latest database entry. Since this forms a chain, it needs to be synchronous. Solving this problem would allow usage of the solution without worrying over the scaling, greatly increasing the quality of the solution.

Another problem that arose is that of how and when to validate the checksums. In theory this is easy but for a large database this can take a lot of time. Along with this there’s the problem of the database not hindering from updating the database (assuming triggers are off). The updates are discovered when validating, but before that unnoticed. As mentioned in related work, the blockchain technique solves this.
References


