Master of Science in Engineering: Computer Security
June 2019

Secure handling of encryption keys for small businesses

A comparative study of key management systems

Jacob Gustafsson
Adam Törnkvist

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

Contact Information:
Authors:
Jacob Gustafsson
E-mail: jagu13@student.bth.se

Adam Törnkvist
E-mail: adtc14@student.bth.se

University advisor:
Dr Fredrik Erlandsson
Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet	:	www.bth.se
Phone	:	+46 455 38 50 00
Fax	:	+46 455 38 50 57

Abstract

Background: A recent study shows that key management in the cooperate world is very painful due to, among other reasons, a lack of knowledge and resources. Instead, some companies embed the encryption keys and other software secrets directly in the source code for the application that uses them, introducing the risk of exposing the secrets. Today, there are multiple systems for managing keys. However, it can be hard to pick a suitable one.
Objectives: The objectives of the thesis are to identify available key management systems for securing secrets in software, evaluate their eligibility to be used by small businesses based on various attributes and recommend a best practice to configure the most suited system for managing software secrets.
Methods: Key management systems are identified through an extensive search, using both scientific and non-scientific search engines. Identified key management systems were compared against a set of requirements created from a small business perspective. The systems that fulfilled the requirements were implemented and comprehensively evaluated through SWOT analyses based on various attributes. Each system was then scored and compared against each other based on these attributes. Lastly, a best practice guide for the most suitable key management system was established.
Results: During the thesis, a total of 54 key management systems were identified with various features and purposes. Out of these 54 systems, five key management systems were comprehensively compared. These were Pinterest Knox, Hashicorp Vault, Square Keywhiz, OpenStack Barbican, and Cyberark Conjur. Out of these five, Hachicorp Vault was deemed to be the most suitable system for small businesses.
Conclusions: There is currently a broad selection of key management systems available. The quality, price, and intended use of these vary, which makes it time-consuming to identify the system that is best suitable based on the needs. The thesis concludes Hachicorp Vault to be the most suitable system based on the needs presented. However, the thesis can also be used by businesses with other needs as a guideline to aid the problem of choosing a key management system.

Keywords: Key management system, encryption keys, cryptography, comparison

Sammanfattning

Bakgrund. En ny studie visar att nyckelhantering i företagsvärlden är väldigt omständligt, bland annat på grund av brist av kunskap och resurser. Istället väljer vissa företag att inkludera krypteringsnycklar och andra mjukvaruhemligheter direkt i källkoden för applikationen som ska använda dem, och därmed introducerar risken att exponera hemligheterna om källkoden skulle bli tillgänglig för en obehörig part.
Syfte. Syftet med denna avhandling är att identifiera tillgängliga nyckelhanteringssystem för att säkra upp mjukvaruhemligheter, bedöma deras lämplighet för småföretag genom att utvärdera dem baserat på olika egenskaper, och rekommendera bästa praxis för att konfigurera det mest lämpliga nyckelhanteringssystemet.
Metod. Nyckelhanteringssystem har identifierats genom en omfattande sökning i både vetenskapliga och icke-vetenskapliga sökmotorer. Identifierade nyckelhanteringssystem jämfördes med ett antal krav skapade från ett småföretags-perspektiv. De systemen som uppfyllde kraven implementerades och utvärderades omfattande genom SWOT analyser baserade på attribut för exempelvis funktioner, prestanda, användarvänlighet och uppskattat framtida stöd. Varje system fick sedan en poäng som jämfördes mot de andra systemen baserat på dessa attributen. Till sist togs även en bästa praxis fram för det mest lämpade nyckelhanteringssystemet.
Resultat. Under avhandlingen identifierades totalt 54 nyckelhanteringssystem med olika funktioner och syften. Utav dessa system jämfördes fem omfattande. Dessa var Pinterest Knox, Hashicorp Vault, Square Keywhiz, OpenStack Barbican och Cyberark Conjur. Utav dessa fem ansågs Hachicorp Vault vara det mest lämpade systemet för småföretag.
Slutsatser. Det finns nuvarande ett brett utbud av nyckelhanteringssystem tillgängliga. Kvalitén, priset och deras syfte varierar vilket gör det tidskrävande att identifiera det systemet som best lämpar sig till ens behov. Avhandlingen anser Hachicorp Vault vara den mest lämpliga baserat på de presenterade behoven, men avhandlingen kan också användas av företag med andra behov som en guide för att underlätta problemet med att välja ett lämpligt nyckelhanteringssystem.

Nyckelord: Nyckelhantering, krypteringsnycklar, kryptering, jämförelse

Acknowledgments

We would like to give a huge thanks to Miljödata AB for their experience and support throughout this thesis, as well as providing us with workspace and necessary hardware and supplies. Our supervisor Dr. Fredrik Erlandsson at Blekinge Institute of Technology also deserves our great gratitude for supporting and guiding us towards the right path and helping us with any question during the thesis.

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface.

CLI Command Line Interface.

HSM Hardware Security Module.

HTTP Hypertext Transfer Protocol.

IRC Internet Relay Chat.

JSON API JavaScript Object Notation API.

KMIP Key Management Interoperability Protocol.

KMS Key Management System.

NDA Non Disclosure Agreement.

PKCS11 Public-Key Cryptography Standard 11.

REST API Representational State Transfer API.

SQL Structured Query Language.

SSH Secure Shell.

SWOT Strengths, Weaknesses, Opportunities, and Threats.

TLS Transport Layer Security.

Chapter 1

Introduction

Today's society relies not only on software but also on the availability of it. Maintaining availability requires among other things confidentiality, integrity, authentication, and non-repudiation which are all obtained using cryptography. Modern cryptography exists using different techniques, however, common for all of them regardless of the algorithm is the need for a secret in some way, often referred to as a key [26]. The term *secrets* can include more than just cryptographic keys. Data such as connection strings, passwords, and API keys are all secrets and equally important in this thesis. Any data that is required by an application but should remain unknown to the outside will be considered a secret in this thesis. The way that secrets are maintained and used is an essential part of security in software development. If secrets are handled incorrectly they might risk exposure, which could lead to services being unavailable due to misuse of API keys or data being stolen, replaced or removed if a database authentication string were to be leaked. Exposed and exploited secrets can result in losses for a company in different ways, for example, by loss of clients due to a damaged reputation or from legal fines. Legal fines are especially relevant after the General Data Protection Regulation (GDPR) was implemented in May 2018. GDPR is a law in the European Union that can be used to punish companies if they handle personal data in an insecure way [15, 50].

One common way of poor secret handling is to embed them in the source code of the project which makes the secrets accessible from wherever the code is accessible. This creates a risk of secrets being leaked from the computer by unauthorized personnel or malicious software [34]. However, this becomes an even greater risk when the code is version-controlled and stored on a centralized local or remote server (e.g., github.com) since the secrets risk exposure from the version-controlled repository as well as from the disk. V. S. Sinha et al. [44] mentions several cases where Amazon AWS secrets have been exposed from Github repositories and used to spin up AWS instances used to mine cryptocurrency. In another case, Luke Chadwick had a pending bill of more than $3,000 for his mistake of making an old private repository public [21]. Implementing a key management system for this, by separating the secrets from the code, not only reduces the risk of exposure but can also include other improvements when handling and maintaining secrets [17].

It is commonly known that cryptography is easy, key management is hard [28]. Since encrypted data is only as secure as its keys [39], key management a crucial part in any security architecture were losing access to the keys implies losing access to the data. Manually managing keys is time-consuming and prone to human errors and misuse, especially when multiple keys have to be managed for different applications and services in the IT infrastructure. This problem can be mitigated by utilizing a key management system (KMS). A KMS manages all keys for an infrastructure, using a single interface, thus minimizing effort and human errors. Because of this, KMSs also allows for scalability previously unfeasible [19, 24].

nCipher Security and Ponemon Institute [35] released a study in April 2019 were 5,856 individuals answered questions about their companies IT security. The respondents are represented across multiple industry sectors in 14 different countries. On the question *How painful is key management on a 10-point scale?* 61% answered seven or higher, suggesting that the respondents find key management very painful [35]. On the year before, 57% answered seven or higher on the same question, suggesting that it is a growing issue. Amongst the top five responses to the multiple choice question *What makes the management of keys so painful?* the following three reasons *Lack of skilled personnel*, *Key management tools are inadequate* and *Insufficient resources (time/money)* could be mitigated if a thorough examination of various key management systems would be available for companies to read [35].

Today, there are multiple key management methods and tools available. However, the cost, complexity, and use case of these solutions vary. Consequently, this creates a process of picking the best fitting one, since some of them might be impractical to implement, a process that is especially painful for a business that lacks the resources to perform such an evaluation. This thesis aims to facilitate this process by comprehensively comparing a set of key management systems, considered suitable for small businesses, where attributes such as accessibility in usage and costs are prioritized higher than security and scalability.

1.1 Aim and objectives

This thesis aims to underline the importance of securely managing secrets, as well as research available software systems to facilitate this. Identified systems will be evaluated based on their use case, performance, implementation, and flexibility when it comes to providing secrets to an application. The evaluation is based on a small business perspective that prioritizes properties like cost, complexity, and required resources above a high security-classification. Lastly this thesis aims to recommend a best practice to securely handle secrets in a way, feasible for small businesses with more limited resources, e.g., budget, knowledge, people, and hardware. The aim can be broken down to the following three objectives:

- To explore what software solutions that are available today to keep secrets secure from unauthorized access.
- To asses their eligibility to be used by small businesses by evaluating the cost and complexity of implementation, simplicity of usage and what problems the solution solves.
- To recommend the best practice to securely handle secrets for small business.

1.2 Research Questions

The following research questions will be answered to achieve the described aim and objectives:

RQ1 What key management systems are currently available to handle software secrets within an application?

RQ2 Based on key management systems from RQ1, how suitable are these for a small business?

RQ3 What is the best practice for implementing and using the most suitable key management system for a small business?

1.3 Limitation

This research will focus on comparing software-based key management systems that facilitate the hassle of maintaining secrets by providing a secure way of storage and an automated way of management. Excluding the section about available key management systems, key management systems that require specialized hardware to achieve better security are not of interest in this report. For example, key management systems that require tamper-resistant hardware to prevent key exposure from a volatile memory dump. This part of key management systems has been excluded due to the cost of implementation and the complexity for testing since this would go against the target audience and the goal for this thesis. Thus, this thesis is not intended for an audience that has a strict security standard to uphold. A comparison of how easy a key management system can be updated, backed up and restored will not be evaluated in this thesis.

1.4 Contribution

The result of this thesis provides a list of key management systems as well as a comprehensive comparative study of a subset of these systems. The list of key management systems is split based on whether the systems are open source or not and includes systems found in various scientific papers as well as in various lists online, included to provide a good idea of available solutions to the reader. The comparative study is based on the business constraints featured in section 4.3.1 at page 16 and aims to provide guidance for small businesses, not requiring a specific security classification but interested in a solution that prioritizes a low cost and ease of use above security. However, neither this study nor its result should be interpreted as a strict guide, but rather as a starting point for conducting a new evaluation based on the needs of the specific business.

1.5 Supporting company

Throughout this thesis, Miljödata AB has given guidance and ideas on how to improve and better form this research. Miljödata AB is a Swedish company, based in

Karlskrona, that has 30 years of experience in software development. They develop web applications that aid human resources managers in their work towards a better working environment for the company.

1.6 Outline

Chapter 2 will cover an introduction to the concept of cryptography and cryptographic key management systems. In Chapter 3 some related works on key management will be summarized and presented. The method used for this thesis and experiment is explained in Chapter 4 together with business constraints, security policy and the questions asked during a SWOT analysis. Moreover, the result of the experiment and answer to the research questions is presented in Chapter 5. An analysis of the results and a discussion about the thesis is available in Chapter 6. Chapter 7 concludes the thesis and also provide possible future work to be done. Lastly, the thesis ends with an appendix with the implementation process and steps for each key management system.

Chapter 2

Background

This chapter will provide an overview and explanation of the relevant concepts and tools mentioned in this thesis.

2.1 Cryptography

Cryptography is the mathematical technique of hiding a string of data or messages from an unwanted party. There are two main operations of cryptography, encryption, and decryption. A key is the main component of cryptography to uphold secrecy if it is kept secure from unauthorized access. When encrypting, a plaintext message and a key are used together in a mathematical function to obfuscate the plaintext into what is called a ciphertext. Reverting the encryption is called decryption and by using a key as well as a mathematical function, similar to encryption. Cryptosystems can also be split into symmetric or asymmetric systems. Symmetric cryptosystems use a symmetric key, meaning that the same key is used in both encrypting and decrypting, while an asymmetric cryptosystem uses different keys for encryption and decryption, called public and private keys. An advantage with an asymmetric cryptosystem is that only the private key needs to be kept secure. The public key is meant to be public and available to anyone who wishes to encrypt data only intended for the party with the private key. This is due to the fact that the data encrypted with a public key only can be decrypted using the private key. Cryptography is used to achieve confidentiality, data integrity, authentication, and non-repudiation [22, 16].

2.2 Cryptographic key management systems

Manually managing keys is a tedious and laborious task were not only the risk of human error increases with each additional key that needs to be secure but also the required time and effort for managing the keys [19]. Since encrypted data is only as secure as the keys used for decrypting, this is not feasible in the long run. A key management system (KMS) can be used to facilitate this problem by providing centralized distribution and storage for all keys used by a company [24]. KMSs exist in multiple different forms, ranging from free smaller applications that run on any regular computer hardware to all-in-one hardware solutions. Simple open source solutions often utilize a regular SQL server for storage, which stores the keys encrypted in a database. However, due to the importance of the key management system, a desirable system should incorporate a hardware security module for handling the keys, or at least provide the option [24].

2.2.1 Hardware security modules

A hardware security module (HSM) can feature special hardware to increase security beyond what is possible with software. HSMs are often validated using a standard called FIPS 140-2, which is defined in four different levels, *Level 1* to *Level 4*. Each level denotes additional security features. The hardware can enable additional security features such as tamper resistance, which enables the device to zeroize or erase all keys in case of a logical or physical attack, or hardware-based randomization which enables True Random Number Generator (TRNG) by generating keys from physical entropy such as thermal noise, avalanche noise, and atmospheric noise. These devices can also feature special processors that can enhance the speed of cryptographic operations by offloading these operations from the application server [34] A HSM can communicate with a KMS using either the PKCS11 or KMIP standard, which both are governed by OASIS [23, 19, 37].

2.2.2 Key operations

A cryptographic key might undergo the following operations: generation, backup, distribution, usage, update, revocation, and deletion. However, every key management system, especially the free ones, might not always support all of these operations. Consequently, it is crucial to know which key operations a system supports before it gets implemented [24].

Generation

The generation operation is the foundation of key management. Cryptographic keys need to be generated securely in an unpredictable way, if not they are vulnerable to multiple attacks, even guessing. Keys should also be generated in a way so that each bit sequence (key) is as likely to occur, and the knowledge of one key should not provide knowledge or clues about other keys generated by the same source. A cryptographic key is generated as a bit sequence from a true-random number generator (TRNG) or a pseudo-random number generator (PRNG). A TRNG utilizes hardware to generate bits from physical entropy such as thermal noise, avalanche noise, and atmospheric noise, compared to PRNG which utilizes a seed to derive random data. One should be careful if this is used when generating a cryptographic key since knowledge of the seed might increase the vulnerability from brute-force attacks [26].

Backup

Regularly backing up data is a critical process for all types of data, especially data crucial for a business. In the event of data loss, a backup enables the data to be restored to its previous state. Different systems and classifications allow for different methods of backup. These range from simple methods that export keys in an encrypted state to an external medium like a USB drive or a CD, to more advanced solutions that utilize special protocols between HSMs. Depending on the business and environment it might also be necessary to implement redundancy. Redundancy is similar to backup with the distinction that it needs to be immediate to allow continuity of the service. Redundancy is achieved through additional hardware, used in case of failure in the primary hardware. For KMSs that utilize HSMs, this means that a secondary HSM with the same keys and configuration is always available for the KMS in case of failure in the primary device [22, 43, 42].

Distribution

Cryptographic keys need to be securely distributed between multiple parties. Cryptomathic, a company with over 30 years of experience in cryptography-based security for businesses, mentions two ways for this. One is a manual way of distribution that utilizes a key encryption key (KEK). The KEK is securely shared manually between parties in advance and later used for encrypting the key to be distributed. This method is also referred to as wrapping since the cryptographic key is wrapped using the KEK (also known as a wrapping key) before transfer to the other party. This method is common for key distribution in the payment domain. However, this method is not securely feasible at a larger scale due to the manual steps included in sharing a KEK, and since each party that receives keys should use its own unique KEK. The other way mentioned utilizes public and private key pairs when distributing the key. By utilizing a key pair, only the party with the private key will be able to decrypt the distributed key. This method replaces the steps of manually sharing a KEK between parties, with the simple process of deploying a public key for each host, which is securely feasible since it can be automated [22, 43, 42].

Usage

As part of the key management life cycle, the usage phase describes the retrieval of an encryption key from the secure storage. During this phase, the entity retrieving a key should undergo authentication and authorization to verify that it is allowed to retrieve that key. The retrieved key should be the active and current version of the keys available [43, 42].

Update

An encryption key might require an update due to numerous reasons. For example, be due to a timed rotation of the key to ensure that no unwanted or expired services have access to a valid key during an extended time, or due to a potential breach or data leak. When a key is updated, the previous version of that key should also be stored in the key management system, making outdated keys accessible if necessary. Performing a rotation of encryption keys might infer issues since all data encrypted with the old key has to be decrypted and then encrypted with the new encryption key [22, 43, 42].

Revocation

If an encryption key should no longer be used, it should be revoked. There are various reasons for key revocation, for example, if it has been compromised and exposed to unauthorized entities, or if it is no longer used actively to perform encryption or decryption. A revoked key could potentially be reactivated again to perform certain decryption actions [22, 43, 42].

Deletion

Key deletion is the last process of a keys lifecycle. A key deletion deletes the key, making it only recoverable from backups, assuming some backup of the key exists. Before deleting a key, it is crucial to know how it has been used since data encrypted with that key can no longer be decrypted once the key is deleted. If a key up for deletion has data associated with it, the key should be securely archived instead to ensure that said data can be recoverable in the future if necessary. [22, 43, 42].

2.3 Tools

During this thesis, five key management systems will be installed, configured, and used to provide a comparison between them. This section will provide a background and explanation of each of these five key management systems: Pinterest Knox, Hashicorp Vault, Square Keywhiz, OpenStack Barbican, and Cyberark Conjur.

2.3.1 Pinterest Knox

Pinterest Knox is a key management system developed by the social media platform Pinterest to solve their problem with managing keys manually and keeping an audit trail. Knox is written in Go and clients communicates with the Knox server using a REST API. Knox's Github repository includes both a server and a client that locally caches keys at the file system. The server and client work together out of the box. However, due to certificates that are public in the repository, the default settings are insecure. Knox is licensed under the Apache License 2.0 agreement [40].

The Knox server communicates using TLS and supports three types of authentication possibilities: Mutual TLS authentication, Github Access Tokens, and the authentication technique SPIFFE[1]. Apart from these, Pinterest also provides some short instructions for implementing a custom authentication provider. The clients communicate with the Knox server through different API endpoints based on the desired functionality, for example, Get Key which enables a machine or a user to retrieve a key from Knox's secure storage. Apart from that one, there are six additional endpoints: one for retrieving the ID of all keys, storing keys, deleting keys, managing the status of keys, updating key versions, and managing the access to keys. For access, Knox supports four types of access structured as a hierarchy: admin, write, read, and none. Admin includes all permissions, write can also read, read is

[1]https://spiffe.io/

self-explanatory and none is to revoke access. These can be assigned to four different types of principal: user, machine, user group, and machine prefix. Knox's audit trail saves information about the action performed, which client performed said action, the IP address of the client, and at what time the action was performed [40].

Per default, Knox uses a volatile temporary database for storing keys. However, a volatile database is not an option for production, and Pinterest has therefore added the options of running the database backend as MySQL or PostgreSQL. Similar to the authentication method, Pinterest also provides some instructions for creating a custom implementation of the key storage backed. To provide confidentiality, Knox encrypts the data stored in the database using AES-GCM with a master encryption key that needs to be accessible by the server. To minimize the manual setup process, Knox is also available as a Docker image [40].

2.3.2 Hashicorp Vault

Vault is a key management system built by the company Hashicorp. A difference between Vault and other presented systems is that Vault is based on a freemium business model, which means that the basic version of Vault is free, but it is also available in paid versions. The paid enterprise versions of Vault includes some additional functionality including, for example, HSM support and a wrapping technique that complies with the FIPS 140-2 standard [8]. The free version is licensed with Mozilla Public Licence 2.0 which allows for commercial use [14].

Vault is written in the programming language Go and available either as source code or as a pre-built binary for different systems, including Windows, macOS, and some Linux distributions. Clients can either communicate with Vault directly through the REST API or via an agent that runs as a daemon on the client machine. The agent supports client-side caching and what Hachicorp calls Auto-Auth, which allows for easy authentication in different environments. Apart from managing secrets, Vaults REST API can also be used for a variety of functions such as configuring Vault, enabling different types of authentication and update access control policies. These functions can also be accessed directly through the Vault binary on the server side. In Vaults documentation, Hachicorp mentions 14 different ways for authenticating the login, ranging from different cloud service providers to certificates, tokens or standard credentials with a username and password. Regardless of the authentication method used for login, Vault will generate a time-based token used for authenticating further requests during its active time. Similar to the authentication possibilities, a wide variety of storage backends are also supported. These are for example cloud services from Amazon, Microsoft or Google, or local services like the local file system or a local database. For audit services, Vault supports logging to a local file, a Syslog server, or directly to a socket. Vault logs information about the client that performed an action, the clients IP address, the action, and at what time it was performed. It is worth noting that if Vault can not audit for some reason, it will not complete any requests until it can write again [13].

The secrets stored in Vault can be stored within different API paths, making it easier to categorize the secrets. Vault also supports access based on these categories which make it easy to manage how the categories can be accessed. For some types of services, for example, databases, Amazon Web Service and SSH, Vault is able to create dynamic credentials that expire after a set period of time. If this is utilized, the real credentials for these services do not have to leave the premise of Vault, making it more secure since only the time-based dynamic credentials can leak [13].

2.3.3 Square Keywhiz

Square Keywhiz is a Java-based application made by the company Square for distributing and managing secrets[46]. Secrets are accessed and managed via a JSON API on the Keywhiz server. Clients can interact directly with this API or via a client provided by Square, similar to how the previous key managers also provided clients.

Keywhiz utilizes two types of users: administrators and clients. Administrators are added before the server starts through a server command and controls the system. Administrators can log in through the CLI using a username and password to manage clients, groups, and secrets in Keywhiz. Clients are added by administrators and utilize client certificates for authentication to the API. Once a client is authenticated, it can add, delete, or rotate keys through the API. Clients access are handled through groups, where clients and secrets are added to non-hierarchial groups, if both the secret and the client are associated with the same group, the client can access that secret. For audit, Keywhiz uses the web server log to perform an audit trail, including which IP address that requested a secret and at what time [47].

By default, Keywhiz communicates over TLS with pre-generated server certificates. On setup, the storage backend can be configured to be either a volatile in-memory database called H2 or a non-volatile database MySQL. Note that H2 should not be used in productions since it stores keys in volatile memory. Keywhiz is licensed with Apache License 2.0 and can be deployed through Docker [46, 47, 48].

2.3.4 OpenStack Barbican

Barbican is a REST API key manager developed by a variety of collaborating parties as an extension of the OpenStack suite. Barbican comes as an API server that is easily installed via a package manager [10]. Apart from the server, OpenStack also provides a client CLI and a python library [7] for interacting with Barbican. Both the client CLI and API can be used to store, update, and retrieve keys; update their access control lists; and also categorize the keys if necessary. Barbican uses the OpenStack module Keystone to perform authentication and can handle a vast variety of authentication methods, including SAML, username and password, tokens and mutual TLS. Barbican keeps an audit trail about the IP address that accessed a certain secret at a specific time but lacks any information about the client associated with that IP address. Barbican is licensed under the Apache 2.0 license agreement [6, 38].

By default, Barbican does not communicate over TLS but can be configured during setup. Barbican supports different types of storage backends, such as a database and all storages that supports KMIP or PKCS11, including HSMs. Barbican also support the usage of multiple storage backends at the same time [38].

2.3.5 Cyberark Conjur

Conjur is a key management system developed in Ruby by the company Cyberark. It is licensed under the GNU Lesser General Public License v3.0 and is easiest installed using Docker [3]. If either the quickstart or the tutorial repository is used when installing Conjur via Docker, it will be preconfigured with TLS and a PostgreSQL backend for storage [4, 5].

Conjur allows for two different types of accounts: users and hosts. Users represent a human that can access Conjur, and a host represents a non-human user. Users can be assigned into groups and hosts can be assigned into what Cyberark calls layers, a type of group for non-human users. This distinction is made to reduce overhead when managing users and non-human users, eliminate errors, and facilitate auditing [12, 9, 2]. Conjurs authentication is based on credentials for the initial request and expiring authentication tokens, renewed every eighth minute, for subsequent requests. Users authenticate using their username and either their password or API key. For hosts, this is done using the host ID and its API key [1]. Users, hosts, groups, layers, and secrets, as well as how they are connected are added through security policy files written in YAML, a language that is readable for both machines and humans. The audit log of Conjur logs information about the client that performed the action, what IP address the client has, what the action was and at what time it was performed [11, 25].

Chapter 3
Related Works

The following chapter will mention three related works that were identified during the literature review. The first article [20] identifies and briefly compares a variety of methods to handle secrets in cloud data storage. The second article [31] performs a comparison of entire security management infrastructures, where key management is one of nine sectors of a security management infrastructure. The third article [45] presents an approach of how to extend the security and functionality of the key management system OpenStack Barbican.

3.1 Key management methods and comparison

A. R. Buchade and R. Ingle [20] performed a research study in 2014 with an objective to compare methods for handling key management in cloud computing. They identified available methods for generating, distributing, revoking, and recovering encryption keys. A. R. Buchadeand and R. Ingle were able to identify five different methods: *managing keys at client side*, *managing keys at cloud service provider side*, *managing keys at both sides*, *managing keys at a centralized server*, and lastly by using a *key splitting technique.*

Managing the encryption keys at the client side means that every client device has a key to access the data stored on the cloud, and it is up to each client to keep the key secure. The identified cloud method uses an asymmetric approach where a Cloud Service Provider (CSP) stores public keys for customers, and the customers themselves store the private key. Managing keys at both the client side and the CSP side splits the key into two subkeys, where both are needed to decrypt the data. Managing keys at a centralized server also take an asymmetric approach with a public key stored at a centralized server for encryption, and a private key stored at users for decryption. The key splitting approach divides the encryption key into n a number of subkeys shared between n clients in the system. Encrypting or decrypting the data requires k out of n subkeys.

The methods identified by R. Buchade and R. Ingle are briefly compared to each other using the properties scalability, security, fault tolerance as well as cryptographic algorithm that best suits the method, i.e., symmetric or asymmetric. In the article, A. R. Buchade and R. Ingle define different cloud computing scenarios and theoretically explains which identified the method that is best suited for each scenario, as well as which identified the method that best suits different applications. As future work,

R. Buchade and R. Ingle mention a practical implementation of the key management methods in a cloud computing environment [20].

3.2 Security management approaches for clouds

In the article, *Security Management interoperability challenges for Collaborative Clouds*, M. Kretzschmar and S. Hanigk [31] compare and evaluate a variety of security management approaches for cloud computing. To identify relevant components, requirements, and interfaces a collaborative scenario between private and public sectors is defined. Requirements for a unified and collaborative cloud security management, are analyzed from four different views: security management functions, collaboration, integration of security management objects, and general requirements. Consequently, A cloud security model is presented, serving as guidelines for implementing and designing a cloud security management system. The paper evaluates fourteen management systems available at the time based on criteria for system design and functionality, where functionality is properties defined as components of security management infrastructure. The systems evaluated are presented in a table based on if a criterion is fulfilled, partially fulfilled, or not fulfilled. The results differ based on the system but common for all is a lack of least support for metadata management [31].

3.3 External key manager for OpenStack Barbican

D. Sitaram et al. [45] present in the article *Standards based Integration of advanced key management capabilities with OpenStack* a practical implementation of the key management system OpenStack Barbican and IBM Security Key Life Cycle Manager. This research aims to improve the security and functionality of OpenStack Barbican by implementing an external key manager via the standardized protocol KMIP. To create this environment D.Sitaram et al. utilize an open source Python module called PyKMIP between Barbican's secret store backend and IBM security key lifecycle manager. PyKMIP functions as a KMIP proxy server which translates incoming Python calls to KMIP communication, supporting at the time version 1.1 of KMIP. The report features this setup process as a guide with some additional notes. D. Sitaram et al. conclude that that is it possible and advantageous to extend the security and functionality of key management in OpenStack by using an external key manager in addition to Barbican since this improves security and adds key management functions that were not present before as well as a user-friendly web interface [45].

Chapter 4
Method

This chapter will explain what methodology was used during the thesis to research and perform an experiment to reach the outlined aim and objectives and to answer the defined research questions.

4.1 Literature review

To create a solid foundation for research and eventually answer the thesis questions, it was necessary to find related work created by researchers in the field. Finding said work also gives an understanding of their point of view and how they approached the issue. Related research was found by searching scientific databases (i.e., Google Scholar) using various search queries to find articles and thesis works that touch the same subject. These findings were evaluated based on relevance by their title, abstract and introduction. If the paper were considered relevant, it was used in a starter set for a Snowballing methodology [53]. This methodology was used in two iterations, allowing us to find additional relevant papers. These papers were evaluated with the same criteria mentioned for the starter set. The result was a set of relevant papers, used in the chapters for introduction, background, and related works.

4.2 Identification process

It was estimated that a couple of key management systems would be found through research papers during the literature review. However, only a couple of articles mentioned examples of key management systems. This forced the search to be expanded to include non-scientific search engines. By searching on the search engine Google a few listing were found that included key management systems. By going through the items in the listings a few more listings were found with more key management tools published. The findings from the scientific papers and the non-scientific listing were extracted into the list of identified key management systems.

4.3 Requirements of a key management system

The following business constraints and security policy were crafted in collaboration with Miljödata. The business constraints were made to simulate how a small software company might handle their business when it comes to their infrastructure and development of software. The security policy defines the rules required to follow, to maintain a certain security level of the infrastructure. The business constraints and security policy were used to provide a set of requirements a key management system have to fulfill to be interesting in this thesis.

4.3.1 Business constraints

The fictive company, referred to as *Fictive inc.*, is a small software development business based in Sweden with between 10 to 49 employees. Fictive inc. develops and maintains a set of web applications that are hosted by Fictive inc. in a virtualized server. Each web application requires different sets of secrets, such as cryptographic keys, credentials, certificates and API keys for external services. Currently, these secrets are saved in a configuration file for each web application. Fictive inc. is in need of a centralized KMS since it allows for key management in a way that previously was unfeasible. Fictive inc. is looking for a KMS that is open source and free of charge due to a tight budget. It does not have to support any specific security standards.

4.3.2 Security policy

- All systems are required to keep an audit trail.
- All systems are required to always communicate over a secure channel, using an approved standard.
- All access to systems is required to be authenticated, preferably through client certificates.
- All access to systems are required to be authorized using an access control list.
- All data and confidential information are required to be kept on-premises.

4.3.3 Requirements of key management systems

The following list defines the requirements Fictive inc. has on a key management system (KMS):

- The KMS shall be able to communicate using a REST API.
- The KMS shall be open source and free of charge, including any third party dependencies.
- The KMS shall log information regarding key operations.
- The KMS shall log information regarding logged in users and actions performed by them.

- The KMS shall log information regarding how a key is accessed when a key is accessed, and by whom a key is accessed.
- The KMS shall encrypt all communication to and from the KMS, using an approved standard.
- The KMS shall enforce secure authentication between clients and itself, preferably using certificates.
- The KMS shall keep access to all keys after a reboot. No keys can be stored in run time memory alone.
- The KMS shall be able to restore all keys from a backup as a fail-safe mechanism.
- The KMS shall keep all keys locally on the premises.
- The KMS shall support an access control list policy, so access to keys can be limited based on user/client.
- The KMS shall allow for scalability. The KMS shall not enforce any limits on the number of keys, users and connected clients.

4.4 Filtering process

Due to the time constraints of this thesis, it was not feasible to comprehensively compare all identified key management systems. Therefore, only a general comparison between all the key management systems was performed. This was done by identifying the features of each key management system and compare them to the requirements of a key management system defined in Section 4.3.3. In this way, it was possible to filter out key management systems that did not fit the defined criteria for a small business. Left was a subset of key management systems that was comprehensively compared.

4.5 Evaluation process

The selected subset of solutions was thoroughly examined using a SWOT analysis to recognize their strengths, weaknesses, opportunities, and threats. The SWOT analysis examines each key management system against a set of attributes. These attributes can be found in the upcoming section, and have been crafted in collaboration with Miljödata AB to get a real-life perspective.

When security changes are made to an environment, it is crucial to also look at the trade-offs that come with the changes. To make an example, a KMS will generally improve the security of an application, but if it interferes too much with productivity and requires too many resources, it is not a fitting solution. Consequently, the implementation process needs to be taken into consideration during the evaluation process. To test the implementation process, each KMS was implemented in a C#

test application. The test application is made as a console application with unit tests to verify secrets from the implemented KMS. More information about this project can be found in Section 4.5.2.

The steps necessary to reproduce the implementation were documented and evaluated in order to measure the complexity of the implementation, performance, and usability. The complexity of implementation was defined as the required time, steps, knowledge needed to completely implement the solution, and if it required changes of the code base. The performance was measured based on the time it took to acquire a secret from the secure storage of the solution evaluated. To evaluate usability, the steps required for using the solution in practice, after implementation, were documented and counted. These steps included how to safely store a secret and also how to acquire it from secure storage. Another way of measuring usability could have been to perform a survey and ask people, using each solution, what they think of it. However, performing a survey on each solution would have been too time-consuming to be feasible during this research, thus it was deselected.

4.5.1 SWOT analysis

The SWOT analysis enables the KMSs to be compared by evaluating a list of attributes for each KMS. The attributes are given a base score of either 1 or -1 depending on if it is judged to be positive or negative. An attribute is considered positive if it is either a strength or opportunity and negative it is either a weakness or threat. The base score is multiplied with a weight based on how crucial an attribute is. An attributes weight is scored ranging from 1 point to 3 points. 1 point represents the least crucial attributes, and 3 represents the most crucial attributes. A KMS's total score is given by adding the scores from each attribute. Evaluated attributes are listed below:

- **Internal (Strengths, Weaknesses):**

    1. **(3 points):** Compared to the other evaluated KMSs, does the KMS include additional features?

    A KMS features, or rather how a KMS can be used is perhaps the most crucial attribute of a KMS, where additional features might be deciding for a user. Likewise, a lack of some specific features could also be deciding. Therefore a KMS needs to be evaluated based on its functionality. However, to keep this attribute feasible, the KMSs will only be compared against the other KMSs up for evaluation, meaning that this attribute is considered positive if the KMS contains additional features compared to the rest of the evaluated KMSs, and negative if the KMS lacks features that the others provide.

    2. **(3 points):** Compared to the other evaluated KMSs, is the response time below average?
    3. **(3 points):** Compared to the other evaluated KMSs, is the network usage below average?

Depending on how the KMS is used, performance can be more or less crucial. Performance is always important to some extent since better performance leads to a smaller negative impact from the KMS. In this thesis, performance is measured as time to retrieve a secret from the KMS, called response time; as well as the network traffic required. For a KMS that is rarely used, for example, if only one application uses it for a couple of hundred requests a day, these are not that impactful for either the application or the network. However, when multiple applications make thousands of requests a day, small differences will add up and be noticeable. These attributes with response time and network usage will be considered positive if the KMS has a response time lower than average and if the network usage is lower than average, respectively.

4. **(3 points):** Does the KMS utilize a secure storage method?

Due to the importance of a KMS, secrets need to be stored secure and recoverable. A secure storage method mitigates attacks against the storage medium, and a recoverable storage method allows the secrets to being restored in case of, for example, hardware failure. A secure storage method is achieved by encrypting the secrets and storing them using a stable storage backend that can be backed up, for example, a database. How secrets are stored and encrypted will probably differ between KMSs. However, the KMS should utilize a master key in some way to ease the recoverability, since such a method will allow recovery from a backup of the storage medium and the master key. If the secrets are encrypted using different keys, all keys need to be backed up, thus making it unfeasible. The KMS should also support external storage since it can facilitate the backup process by backing up medium using methods already in place for backing up databases and disks for example. External storage also allows the master key and storage to be hosted at different machines.

5. **(3 points):** Does the KMS provide a sufficient audit log?

A sufficient audit log is a must for any critical system, especially a KMS. An audit log is required to hold users accountable for their actions but can also be used to, for example, statistics. A sufficient audit log should include the action, e.g., fetching, creating, modifying or removing a key to name a few; a timestamp of when said action was performed; and by whom said the action was performed.

6. **(3 points):** Does the KMS provide access control?

Access control, in a KMS, enables different users to have access to different secrets. Access control is useful for centralizing key management since different applications can be configured to use different KMS users. Hence, an application can only access the secrets relevant to that application.

7. **(2 points):** Does the KMS support multiple methods of authenticating?

The purpose of authenticating is to prevent unauthorized users from accessing the KMS. By allowing multiple methods of authentication, a KMS can be applicable in a wider range of scenarios.

8. **(2 points):** Does the KMS natively start automatically?

Whether a KMS should automatically start can be argued. It can be seen as positive since it does not require any human input. It can also be seen as negative since it means that the KMS needs to save all credentials used for accessing the stored secrets to the disk. For this evaluation, it is considered a good thing if the KMS can automatically start since the evaluation is focused more on usability with KMSs than security.

9. **(2 points):** Does the KMS's manufacturer provide well-written documentation?

Well-written documentation can save time when implementing a KMS. For documentation to be considered well-written in this thesis, it should at least include steps for installation and use. These instructions should be easy to follow and lead to the result desired.

10. **(2 points):** Does the KMS support an HSM?

A KMS's security can be significantly upgraded using an HSM, as mention in Section 2.2.1 on page 6. It is therefore beneficial if a KMS has HSM support. HSM support is also good to future proof a KMS since the KMS does not have to be replaced in the future if an HSM were to be added to the security infrastructure. The level of HSM support may vary between KMSs, therefore as a minimum requirement, the KMS should at least be able to store the master key in the HSM.

11. **(2 points):** Does the KMS allow commercial use?

Using an open source project commercially sometimes differ from using it privately, especially if it is used commercially for profit. It is, therefore, necessary to take into consideration how a KMS is licensed. Even if a KMS is not freely licensed, e.g., Apache License 2.0, Mozilla Public License 2.0 or GNU Lesser General Public License v3.0, to name a few, it might still be possible to obtain a permit or deal. However, the KMS will have to be advantageous in other aspects for it to be worth it.

12. **(2 points):** Is the KMS easy to use?

To have as little impact on productivity as possible, a KMS should be easy to use. Whether a KMS is easy to use or not is, for this thesis, decided by evaluating the code required for an application to utilize the KMS, as well as how easy it is to add and modify secrets. As mention in the limitations section (Section 1.3, on page 3), how easy a system is to update or backup is not tested and verified in this thesis due to time constraint.

13. **(1 point):** Is the KMS easy to install?

A KMS should optimally be easy to install as well as easy to use. Consequently, a KMS should have an evident installation process that is easy to follow. A KMS that requires little or no editing in the code base is considered good and KMS that require extensive editing of the source code is bad.

- **External (Opportunities, Threats):**

14. **(3 points):** Does the KMS have any known security vulnerabilities that are not likely to be fixed?

A KMS is mainly used for security and convenience. If a KMS contains any known security vulnerabilities not likely to be fixed, these are both threatened. The security is threatened due to vulnerabilities, and the convenience is threatened since the KMS can no longer be trusted. Therefore, a KMS should have no known vulnerability for its latest version. Verifying known vulnerabilities is done through CVE Details[1], as well as the issue section on a KMS's Github page. Found vulnerabilities should either be fixed or addressed. If a vulnerability is recently found, during the last week, an estimation will be made whether it is likely to be fixed or not. Such an estimation will be based around the activity of a KMS's Github repository.

15. **(3 points):** Compared to the other evaluated KMSs, is the KMS's ratio of open issues relative to closed issues below average?

The goal of measuring open issues relative to closed issues is to get a guideline about how much impact the community has on the KMS. This ratio should optimally be 0. However, this would mean that there are no open issues, which is unreasonable for a well used KMS. Even more so since open issues contain more than just bugs, subjects like general questions, feature requests, and typos have all been observed as open issues for evaluated KMSs. This attribute is measured positive or negative by comparing a KMS ratio to the average ratio. This attribute is considered positive if a KMS's ratio is below the average ratio, and negative if its ratio is above the average ratio.

16. **(3 points):** Is the KMS actively maintained and developed by some party?

A KMS should be actively under development to strengthen the possibility of future updates, for example, bug fixes and additional features. Therefore, it is essential that a party actively works on the KMS. This attribute is considered positive if a commit has been done to the KMS's Github repository in the last month. If the latest commit is older than one month, this attribute is considered negative. This question is similar to the question regarding open and closed issues. However, this question also covers the activity made beyond the tasks in the issues section.

[1]https://www.cvedetails.com/

17. **(2 points):** Does the developer(s) of the KMS provide any type of forum for technical support, in addition to the issues tab on Github?

During the installation or usage of a KMS, it might be necessary with technical support. Technical support can at some extent be provided by submitting an issue to the repository at Github. However, this might not always be the optimal way, which makes a forum outside of Github a great alternative, for example, a Slack or IRC channel. Such forums might also allow for discussions between users rather than between users and developers as seen on Github.

18. **(1 point):** Is the KMS popular among other developers?

A popular KMS with a large userbase is beneficial compared to a less popular one. When a KMS is more popular, more users are around to find bugs in said KMS and found bugs are more likely to be fixed, leading to a more stable KMS with time. More users also mean a larger community, which is beneficial when it comes to for example help, guides, and additional functionality from other sources than the developers. During the evaluation, a KMS popularity is measured by how many times a KMS has been starred or watched on its Github page. This attribute would optimally be measured by the number of installations for a KMS alternatively the number of downloads. However, this information is not publicly available on Github. Consequently, this attribute should be seen as how popular a KMS is compared to the other evaluated KMSs, rather than how well used a specific KMS is.

19. **(1 point):** Does a KMS's source code include unit tests?

An advantage of open source software compared to close source software is the ability to further develop and maintain it in-house. If a KMS were to be further developed in-house, it would be beneficial if unit tests were provided by the original developers. The unit tests could then be used to verify the original code which would save time. Unit tests for the original code are especially useful if the core of the KMS were to be modified at some stage.

20. **(1 point):** Can the KMS's functionality be easily extended?

This attribute focuses on whether the KMS is developed with easy extendability in mind. Easy extendability is hard to define concrete but includes, for example, modular designs were parts of a KMS can be easily replaced or support for plugins that can extend a KMS's functionality.

4.5.2 Experimental environment

As mentioned previously, trade-offs are expected when making security changes. Thus, an experimental environment was used to evaluate the installation process and user experience of selected KMSs. The experimental environment was designed in collaboration with Miljödata AB, to verify that it, at least, evaluates their needs and use case. It consisted of the KMS to be evaluated and a console application that interacted with it.

Console application

The console application was designed to evaluate how an application would interact with the KMS. KMSs evaluated in this thesis uses REST APIs for communications between the application and the KMS. The applications REST API calls were all made in a similar way to each other, but with different data based on the calls. For example, fetching the key might be done with a GET-request to the KMS with the name of the key, and storing a key might be done by using a POST-request with a key and some information about it. Due to similarities between different REST API calls, the console application was focused on evaluating how an application would initiate and authenticate the communication with the specific KMS, rather than the features of a specific KMS. Consequently, only one type of REST call was implemented, a REST call that fetches a key in the form of a string. The key was verified using unit tests, and the execution performance was benchmarked using a package called BenchmarkDotNet[2]. Additionally, the network traffic was measured using Process Monitor to monitor all traffic being sent between the test application and the KMS.

The console applications utilize a factory pattern to create objects for different KMS clients. The factory pattern creates new objects for each KMS which shares a common interface. The interface contains methods for creating a specific KMS client implementations as well as a function to get a key based on a given ID. A KMS's client was implemented as a class that can authenticate and connect to the KMS through a secure method, most commonly using certificates, as well as retrieving a key based on a given ID. Once a key is retrieved, it is verified through the interface using unit tests. The source code of this console application can be found at Miljödata's Github account[3].

KMS environment

The KMS environment was used for testing the server side of each KMS. The environment consists of a virtual machine running Ubuntu 18.04[4] in Oracle VM Virtualbox[5]. The virtualized hardware in this machine was running with 1 CPU, allocated with 4096 GB RAM and 60 GB storage. The virtual environment was chosen due to the flexibility it provides when using snapshots. A snapshot is a machine state saved to the disk, which can later be restored. The environment utilizes a snapshot for each KMS when it was successfully implemented, as well as a snapshot taken when the Ubuntu machine has been fully configured, called a based snapshot. To keep the evaluation and the benchmarked tests as fair as possible, each KMS needed to be installed on a clean machine. Utilizing snapshots saved time when evaluating the key management systems since the base snapshot could be used as a starting point instead of having to install Ubuntu for each new KMS to be implemented.

[2]https://benchmarkdotnet.org
[3]https://github.com/Miljodata/public-KeyManagementPoC
[4]http://releases.ubuntu.com/18.04/
[5]https://www.virtualbox.org/

Each KMS was installed using the most hands-on and basic way that has been documented. For example, if a KMS has documentation for a hands-on step-by-step installation process and a Docker installation, the step-by-step method was chosen. If Docker was the only documented way, which is the case with Conjur, that method was chosen. By installing the KMSs using the most hands-on way documented, the upcoming sections about installation could be written as a worst-case scenario, assuming that no errors or unexpected messages occur. The reader should therefore not expect a more complicated installation process than the one documented in this thesis. Generally, the more hands-on a KMS's installation and configuration process are, the more transparent are its dependencies and settings. A business might, therefore, prefer a more hands-on process than a Docker container.

Performance analysis

After the five KMSs to be evaluated were installed and configured in separate snapshots of the virtualized Ubuntu environment, a script was created to automate the performance testing. The script changed the configuration file for the test application, switched snapshot, booted the virtual machine, ran the unit tests and the benchmarking, and then saved the results. The performance analysis was executed on a Lenovo T580 laptop with an i7-8550U CPU at 1.80 GHz and 16 GB RAM running Windows 10 64-bit. During the performance analysis, other programs were closed to minimize the impact of other processes running on the system. Therefore, only the necessary system processes, VirtualBox, Process Monitor, and the test application were running during the performance analysis. During the performance analysis, Process Monitor was used for measuring each KMSs impact on the network by measuring the network traffic to and from the test application. After the performance analysis, the results were compared, and the average time and network impact were calculated for each KMS.

4.6 Validity

4.6.1 Construct validity

To fulfill the aim of identifying different KMSs, comparing them based on their suitability for small business, and lastly presenting a KMS that is best suited for a small business, the methodology was constructed in collaboration with Miljödata AB, a small company experienced in IT and web applications. After discussing KMSs with Miljödata AB, it was clear that the project needed to focus on identifying different KMSs and evaluate them based on their suitability for Miljödata, or another company with similar infrastructure. With the use of our academic experience, their need for a suitable KMS was implemented as business constraints, a security policy, and a SWOT analysis. The business constraints and the security policy allows unsuited KMSs to be quickly filtered, and thus, the scored SWOT analysis could be more in depth in the same timeframe. By utilizing a SWOT analysis, suited KMSs could be examined beyond what would have been possible from just analyzing their documentation. Thus, the result presents the most suitable KMS instead of a list with suitable KMSs. The results also contain a best practice guide about how that specific KMS should be securely configurated for production.

4.6.2 Internal validity

Together with Miljödata's experience in IT and web application development, the test environment and evaluation were made to be as relevant as possible. Both in terms of how the environment is structured and how KMSs are implemented, but also in terms of what attributes are deemed necessary to evaluate and how they should be prioritized.

The collaboration with Miljödata could also be considered a disadvantage since the experiment and evaluation are not formed based on opinions from multiple small businesses. Therefore, it is possible that other entities in the field would disagree regarding the construction of the experiment, the attributes under evaluation, or the attributes priority. A survey targeting multiple small businesses could diversify this. However, due to time constraints and lack of connections to the target group, this was not pursued further.

4.6.3 External validity

As mention in the previous sections, the method, or at least the experimental environment and the evaluation, is influenced by Miljödata AB. Consequently, the result will be targeted to small businesses that have the same needs as those defined in the business constraints and the security policy. The result presented is always applicable to businesses that don't require a specific security classification. However, due to some of the business constraints or some of the items in the security policy, for example, the requirement of an open source KMS free of charge or the requirement of local storage. Other businesses might have preferred an evaluation between five other KMSs instead. Business with other needs, preferences, or infrastructure might, therefore, find more optimal solutions in KMSs not evaluated in this thesis. A clear example of this could be if they use a cloud service provider that also offers a KMS, such as Amazon, Google or Microsoft; or if they prefer to pay for a KMS service to avoid the responsibility that comes with handling secrets, for example, availability and backups.

4.6.4 Reliability

A few different actions were taken to get the results of the performance tests as reliable and fair as possible. The first action made was to only run the necessary system services on the computer with addition to the necessary applications for the execution of the tests. Secondly, the performance tests were started at the end of the workweek and left running during the weekend to minimize possible human interactions with the system. Thirdly, the benchmarking framework used for the performance tests has built-in functionality, by default, to reach the best possible precision of each of its test [18]. Lastly, ten tests were conducted on each of the six implemented functions in the test project, every time the test project was executed. From the test script, the test project was executed ten times per key management system. Eventually, the average execution time of each key management system was calculated based on 600 results from the performance tests.

The SWOT analysis performed on each key management system has a defined structure of what is being evaluated and what is considered sufficient or insufficient in the comparison. By defining and presenting the guidelines that are followed for the SWOT analysis, an insight is given of how the key management systems have been evaluated. This also helps mitigate possible bias the researches might get about certain key management systems and still keep the comparison objectively.

Chapter 5

Results

This chapter shows the findings during this research divided into a section for each research question. The first section shows the identified KMSs, the second shows a comparison of the identified systems and lastly a best practice guide is defined.

5.1 Research question 1

The first research question strives for getting a general idea of what KMSs that are available, at the moment, for handling software secrets. The list of identified KMSs are based on the findings from scientific papers [27, 51, 45] as well as various listings [32, 52, 41, 49, 36]. A total amount of 54 KMSs were identified with various aims and features. These are presented in Table 5.1 and Table 5.2, divided into open source and closed source respectively.

Table 5.1: List of identified open source key management systems

Key Management System	**Website**
Ansible Vault	https://docs.ansible.com/ansible/2.4/vault.html
Chef Vault	https://github.com/chef/chef-vault/
Pinterest Knox	https://github.com/pinterest/knox
Poise Citadel	https://github.com/poise/citadel
Lyft Confidant	https://github.com/lyft/confidant
Hashicorp Vault	https://github.com/hashicorp/vault
XOR Data Exchange Crypt	https://github.com/xordataexchange/crypt
Shopify EJSON	https://github.com/Shopify/ejson
Square Keywhiz	https://github.com/square/keywhiz
OpenStack Barbican	https://github.com/openstack/barbican
Cloudflare Red October	https://github.com/cloudflare/redoctober
Oleiade Trousseau	https://github.com/oleiade/trousseau
Bastillion-io Bastillion	https://github.com/bastillion-io/Bastillion
Neat S.r.l. Kmc-Subset137	https://github.com/neatsrl/Kmc-Subset137
PrivacyIDEA	https://github.com/privacyidea/privacyidea
Flix- Keeto	https://github.com/flix-/keeto
Cyberark Conjur	https://github.com/cyberark/conjur
Docker Secrets	https://docs.docker.com/engine/swarm/secrets/
Manifold Torus	https://github.com/manifoldco/torus-cli
Mozilla SOPS	https://github.com/mozilla/sops
Fugue CredStash	https://github.com/fugue/credstash
Codahale Sneaker	https://github.com/codahale/sneaker
Meltwater Secretary	https://github.com/meltwater/secretary
Shyiko Kubesec	https://github.com/shyiko/kubesec
Latchset Custodia	https://github.com/latchset/custodia
FreeIPA	https://github.com/freeipa/freeipa
T-Mobile T-Vault	https://github.com/tmobile/t-vault
EnvKey	https://github.com/envkey/envkey-source
Schibsted Strongbox	https://github.com/schibsted/strongbox
Dcoker Biscuit	https://github.com/dcoker/biscuit

Table 5.2: List of identified closed source key management systems

Key Management System	**Website**
Amazon AWS Secrets Manager	https://aws.amazon.com/secrets-manager/
Amazon AWS KMS	https://aws.amazon.com/kms/
AppViewX CERT+	https://www.appviewx.com/products/cert/
Cryptomathic Crypto Key Management System	https://www.cryptomathic.com/products/key-management/crypto-key-management-system
Fornetix Key Orchestration	https://www.fornetix.com/our-products/
Futurex Key Management Servers	https://www.futurex.com/products/category/key-management-servers
IBM Security Key Lifecycle Manager	https://www.ibm.com/us-en/marketplace/ibm-security-key-lifecycle-manager
KeyNexus Key Management as a Service	https://keynexus.net/solutions-ukm/key-management-as-a-service-kmaas/
Microsoft Azure Key Vault	https://azure.microsoft.com/services/key-vault/
Google Cloud KMS	https://cloud.google.com/kms/
Oracle Key Vault	https://www.oracle.com/database/ technologies/security/key-vault.html
Oracle Key Manager	https://www.oracle.com/storage/tape-storage/key-manager-3/
Quintessence qCrypt	http://www.quintessencelabs.com/products/encryption-key-management/
Gemalto Safenet Virtual KeySecure	https://safenet.gemalto.com/data-encryption/enterprise-key-management/virtual-key-secure/
SSH.com Universal SSH Key Manager	https://www.ssh.com/products/universal-ssh-key-manager/
Thales Vormetric Data Security Manager	https://www.thalesesecurity.com/products/data-encryption/vormetric-data-securitymanager
Townsend Secuirty Centralized Encryption Key Management Server	https://www.townsendsecurity.com/products/centralized-encryption-key-management
HancomSecure KeyManager	https://www.hsecure.co.kr/en/solutions/solution_list.php?cate_cd=0104
Kryptus KNET	http://www.kryptus.com/en/knet
Unbound Key Control	https://www.unboundtech.com/product/unbound-key-control/
Bloombase KeyCastle	https://bloombase.com/products/spitfire/keycastle
CipherCloud Key Management	https://www.ciphercloud.com/key-management
Hytrust KeyControl	https://www.hytrust.com/products/keycontrol
Zettaset Xcrypt	https://www.zettaset.com/products/encryption-key-management

5.2 Research question 2

The second research question strives for a more in-depth view of how well the identified key management systems would fit a small business based on the business constraints and security policy outlined. As shown in 5.1, a vast amount of tools for key management were identified. It would be interesting to cover and compare all identified tools comprehensively but time and manpower restrict this, thus only five tools will be comprehensively compared. Table 5.3 shows a general comparison of the features of all identified KMS with respect to the requirements outlined for a KMS.

Table 5.3: A comparison of all identified KMSs with respect to the requirements of a KMS. Empty cells means *Yes*.

	REST API	**Free / Free third party cost**	**Audit log**	**Secure communication**	**Authenticate with certificate**	**Keys kept on-premises**	**Access control list**
Ansible Vault	No		No	N/A	No		No
Chef Vault	No		No	N/A	No		
Poise Citadel	No	No	No		No	No	
Pinterest Knox							
Lyft Confidant	No				No		
Hashicorp Vault							
XOR Data Exchange Crypt	No		No				
Shopify EJSON	No		No	N/A	No		
Square Keywhiz							
OpenStack Barbican							
Cloudflare Red October			No		No		
Oleiade Trousseau	No		No	N/A	No		No
Bastillion-io Bastillion	No				No		
Neat S.r.l. Kmc-Subset137	No		N/A	N/A	N/A	N/A	N/A
PrivacyIDEA	No				No		
Flix- Keeto	No			N/A	No		
Cyberark Conjur					No		
Docker Secrets	No				No		No
Manifold Torus	No		No		No	No	
Mozilla SOPS	No	No		N/A	No		
Fugue CredStash	No	No		N/A	No	No	
Codahale Sneaker	No	No			No	No	
Meltwater Secretary	No				No		
Shyiko Kubesec	No			N/A	No		
Latchset Custodia					No		No
FreeIPA	No		No				
T-Mobile T-Vault					No		
EnvKey		No			No	No	
Schibsted Strongbox	No	No			No	No	
Dcoker Biscuit	No	No	N/A	N/A	No		
Amazon AWS Secrets Manager		No				No	
Amazon AWS KMS		No				No	
AppViewX CERT+		No			N/A	N/A	
Cryptomathic Crypto Key Management System		No			N/A	N/A	
Fornetix Key Orchestration		No			N/A	N/A	
Futurex Key Management Servers		No	N/A	N/A	N/A		N/A
IBM Security Key Lifecycle Manager		No	N/A		N/A	N/A	
KeyNexus Key Management as a Service		No			N/A		
Microsoft Azure Key Vault	No	No				No	
Google Cloud KMS		No				No	
Oracle Key Vault	No				N/A		
Oracle Key Manager	No	No	N/A		N/A		
Quintessence qCrypt		No			N/A		
Gemalto Safenet Virtual KeySecure		No			No		
SSH.com Universal SSH KeyManager	No	No			N/A		
Thales Vormetric Data Security Manager		No			N/A		
Townsend Secuirty Centralized Encryption Key Management Server		No			N/A		
Hancom Secure Key Manager		No			N/A		
Kryptus KNET		No	N/A		N/A		
Unbound KeyControl		No			N/A		
Bloombase KeyCastle		No			N/A		
CipherCloud Key Management		No	N/A		N/A	N/A	
Hytrust KeyControl		No			N/A		
Zettaset Xcrypt		No	N/A	N/A	N/A		

All columns in Table 5.3, except *Authenticate with certificate*, are a requirement according to Section 4.3.3. *Authenticate with certificate* are considered preferable but not a requirement. The key management systems that are not supporting REST API, are not free, that do not utilize an audit log or access control list and do not keep the keys on-premises are excluded from this thesis. This means that the key management systems that remain are Pinterest Knox, Hashicorp Vault, Square Keywhiz, OpenStack Barbican, Cyberark Conjur, and T-Mobile T-Vault. However, T-Mobile T-Vault is an extension of Hashicorp Vault and was therefore excluded from a comprehensive comparison. The filtering process concludes into the following five key management systems used in the comprehensive comparison:

1. Pinterest Knox
2. Hashicorp Vault
3. Square Keywhiz
4. OpenStack Barbican
5. Cyberark Conjur

The acquired data from the Github repositories and documentation of each tool that is comprehensively compared is gathered into the Table 5.4. The table represents an overall comparison between the tools with regards to a wide range of aspects, including the quality of the documentation, if it has a sufficient audit log and has any known security issues. The content of Table 5.4 is used in the SWOT analysis in a prioritized order.

Table 5.4: A comparison of a subset of the KMS with respect to the SWOT analysis questions. Empty cells means *Yes*. The data was collected 2019-04-23.

	Knox	**Vault**	**Keywhiz**	**Barbican**	**Conjur**
Include secure storage?					
Sufficient audit log?			No	No	
Include access control?					
Multiple auth methods?	No				No
Boot automatically?		No			
Good documentation?	No		No	No	
Support HSM?	No		No		
Open for commercial use?					
Any known vulnerabilities?	No	No		No	No
Techical support available?	No		No		
Include unit tests?					
Popularity (users)?	756	12826	2151	219	315
Open / Total issues (%)?	0	11.1	31.5	12.5	25.5
Commits latest month?	0	683	6	15	144
Response time (ms/op.)?	0.78	1.56	2.31	83.37	6.95
Network usage (kB/op.)?	3.65	3.93	2.86	1.76	3.63

The attribute *Popularity (users)?* in Table 5.4 shows how popular the compared key management systems are among developers on Github by combining the number of users watching and the number of stars the repository has. The popularity of the KMSs varied greatly. Hashicorp Vault was, by far, the most popular and known key management system at the time of collecting the data. The average amount of users that have watched or starred any of the tools was 3243.4 users, thus only Hashicorp Vault was over the average.

Open / Total issues (%)? shows how well the contributors of the KMS care for solving open issues on Github by dividing the number of open issues with the total amount of issues. Pinterest Knox has the lowest ratio between the KMS with no open issues and Square Keywhiz has the highest ratio of open issues. The average ratio was 16,12%, which Pinterest Knox, Hashicorp Vault and OpenStack Barbican were all under.

The attribute *Commits latest month?* shows how frequent the key management systems are developed by comparing the number of commits the latest month, the period 2019-03-23 - 2019-04-23. Hashicorp Vault was the most actively maintained system and Pinterest Knox had the lowest activity during the latest month with no commits at all. The average number of commits during the period was 169,6 commits which only Hashicorp Vault was over.

The last two attributes *Response time (ms)?* and *Network usage (kB)?* shows the result of the performance during the usage of the key management systems. The first of them shows the average execution time of retrieving a secret from the key management system and the second shows a comparison of the amount of data that is sent over the network when retrieving a secret. The fastest key management system to retrieve a secret was Pinterest Knox and the slowest was OpenStack Barbican. All of the tools except for OpenStack Barbican has a faster execution time than the average time of 18.99 ms. OpenStack Barbican was the one that sent the least data over the network while retrieving a secret from its storage. Hashicorp Vault used the most data to do the same task. Square Keywhiz and OpenStack Barbican sent less data over the network compared to the average amount of 3.17 kB between all the compared key management systems.

The following five tables, 5.5, 5.6, 5.7, 5.8, and 5.9, shows the SWOT analysis performed on each compared key management system respectively, summarizes their abilities and provide a final score. The *ID* column of these tables denotes the questions presented in Section 4.5.1. The *Pts* column denotes the priority each question has.

Table 5.5: The SWOT analysis of Pinterest Knox

PINTEREST KNOX				
	ID	**Pts**	**STRENGTHS**	**WEAKNESSES**
INTERNAL	1	3	Knox can be deployed using Docker for a minimized setup process. Knox has an built in client that can be used to automatically cache keys and keep them up to date.	
	2	3	The response time of Knox is below average.	
	3	3		Knox sends more data over the network than the average.
	4	3	In production Knox uses MySQL or PostgreSQL as the storage backend with encrypted data.	
	5	3	Knox saves information of what action that has been performed, by which client and IP and at what time.	
	6	3	Knox support access control with four levels of privileges.	
	7	2		It's possible to authenticate in three different ways, Mutual TLS, Github Access Tokens and through SPIFFE. However, during the setup process, only Github Access Tokens can be used.
	8	2	Knox can natively start automatically using a command at server boot up.	
	9	2		The documentation explains various functionalities, however it lacks the technical details on how to use the provided API.
	10	2		Knox does not support HMS.
	11	2	Knox can be used for commercial purposes.	
	12	2	Knox can be easily used by implementing it as a client wrapper class.	
	13	1		The installation process requires a lot of editing in the source code to make it production ready.
	ID	**Pts**	**OPPORTUNITIES**	**THREATS**
EXTERNAL	14	3	No security vulnerabilities have been published.	
	15	3	The ratio of open and closed issues are under the mean ratio.	
	16	3		The last month the repository has none commits made, which is under the mean.
	17	2		Pinterest does not provide any technical support.
	18	1		Knox is less popular than the mean popularity.
	19	1	The source code includes unit tests.	
	20	1	It is possible to implement a custom authentication method and storage backend.	
		Score:	(3+3+3+3+3+2+2+2)-(3+2+2+2+1)+(3+3+1+1)-(3+2+1)=13	

Table 5.6: The SWOT analysis of Hashicorp Vault

HASHICORP VAULT				
	ID	**Pts**	**STRENGTHS**	**WEAKNESSES**
INTERNAL	1	3	Allows for categorizing keys and give access to a category. Has dynamic keys that expire after a specific time.	
	2	3	Vault has a lower response time than the average time.	
	3	3		Vault sends more data over the network than the average.
	4	3	Vault support multiple different secure storage backends, such as different databases, cloud storages and file system.	
	5	3	Vault logs information about which client and IP that accessed what key at what time	
	6	3	Hashicorp Vault has access control.	
	7	2	It is possible to authenticate using multiple different types of methods, including Github credentials, tokens, certificates and credentials to cloud providers.	
	8	2		The server can not start automatically, it requires unlocking of the vault using a set of keys.
	9	2	The documentation is well written and has thorough explanations of the features and configurations.	
	10	2	Vault support a HSM as the storage backend.	
	11	2	Vault is allowed to be used in a commercial project.	
	12	2	Vault is easily used in source code by implementing it as an API client.	
	13	1	This KMS only requires creation of configuration files and running commands to be installed.	
	ID	**Pts**	**OPPORTUNITIES**	**THREATS**
EXTERNAL	14	3	No security issues have been published for Vault.	
	15	3	The open / closed issues ratio is below the mean ratio.	
	16	3	The repository has more commits pushed than the average.	
	17	2	Hashicorp offers technical support for Vault through an open IRC channel.	
	18	1	Vault is more popular than the average between the other KMS.	
	19	1	The source code includes unit tests.	
	20	1	It is possible to implement additional custom backends for either authentication or storage as a plugin.	
		Score:	(3+3+3+3+3+2+2+2+2+2+1)-(3+2)+(3+3+3+2+1+1+1)-()=35	

Table 5.7: The SWOT analysis of Square Keywhiz

SQUARE KEYWHIZ				
	ID	**Pts**	**STRENGTHS**	**WEAKNESSES**
INTERNAL	1	3	Supports Docker for smooth deployment.	
	2	3	The response time of Keywhiz is lower than the average	
	3	3	The amount of data sent by Keywhiz is below average.	
	4	3	In production Keywhiz only support MySQL and encrypt the stored data.	
	5	3		Keywhiz uses the web server log to perform an audit trail. This includes which IP that requested which secret at what time, but lacks information regarding which client that requested the secret
	6	3	Keywhiz support access control.	
	7	2	It is possible to authenticate using either password or certificates.	
	8	2	Keywhiz can start automatically if the machine is rebooted.	
	9	2		The documentation is spread out on three different pages and only explains how to use the KMS in development mode but lack configuration guidelines for making it production ready. The documentation also include multiple broken links.
	10	2		This KMS does not support HSM.
	11	2	Keywhiz can be used for commercial purposes.	
	12	2	It is possible to implement Keywhiz as an API client in a ASP.Net project.	
	13	1	Keywhiz only requires running commands to install the server.	
	ID	**Pts**	**OPPORTUNITIES**	**THREATS**
EXTERNAL	14	3		Keywhiz has one security vulnerability publically published and has been open for the last three years [33].
	15	3		Keywhiz has more open issues than the average between the other KMS.
	16	3		The repository has been less active than the average activity.
	17	2		The documentation does not mention how to get support if needed.
	18	1		The popularity of Keywhiz is below the average popularity.
	19	1	The source code include unit tests.	
	20	1	Keywhiz can be extended using a client that exposes the secrets as a filesystem. This removes the requirement of accessing them through the API.	
		Score:	(3+3+3+3+3+2+2+2+2+1)-(3+2+2)+(1+1)-(3+3+3+2+1)=7	

Table 5.8: The SWOT analysis of Openstack Barbican

OPENSTACK BARBICAN				
	ID	**Pts**	**STRENGTHS**	**WEAKNESSES**
I N T E R N A L	1	3	Barbican allows for categorizing keys.	
	2	3		The response time of Barbican is over the average.
	3	3	The amount of data sent over the network is lower than average.	
	4	3	Barbican support different types of secure storage backends including database, KMIP or PKCS11 based storages.	
	5	3		Barbican keeps an audit trail of what IP that accessed a certain secret at a specific time, but lacks information of which client that accessed it.
	6	3	Barbican comes with an access control list.	
	7	2	It is possible to authenticate using multiple methods, such as SAML, user credentials, tokens and certificates.	
	8	2	Barbican can be restarted automatically after a reboot.	
	9	2		The documentation is well written but lacks a logical structure making it hard to find the right section.
	10	2	Barbican can be extended to use HSM for greater security.	
	11	2	Barbican can be used for commercial purposes.	
	12	2	Barbican can easily be implemented in a client wrapper class.	
	13	1	The installation process includes creating a script, editing a configuration file and running commands.	
	ID	**Points**	**OPPORTUNITIES**	**THREATS**
E X T E R N A L	14	3	No security related issues have been found.	
	15	3	The Barbican repository has less open issues than the average.	
	16	3		Barbican has less commits pushed to the repository than the average.
	17	2	OpenStack has an open IRC channel dedicated for technical support of Barbican.	
	18	1		Barbican is not as popular as the average.
	19	1	The source code include unit tests.	
	20	1	It is possible to extend Barbican to use multiple storage backends simultaneously.	
		Score:	(3+3+3+3+2+2+2+2+2+1)-(3+3+2)+(3+3+2+1+1)-(3+1)=21	

Table 5.9: The SWOT analysis of Cyberark Conjur

CYBERARK CONJUR				
	ID	**Pts**	**STRENGTHS**	**WEAKNESSES**
INTERNAL	1	3	Cyberark Conjur comes in a Dockerized container.	
	2	3	Conjur perform better than the average of the KMS tested.	
	3	3		Conjur sends more data than the average over the network.
	4	3	Conjur uses PostgreSQL as the secure storage backend.	
	5	3	The audit log of Conjur logs information of which client and IP that has performed an action, what the action is and when it occured	
	6	3	The KMS support access control.	
	7	2	Allows for authentication using account ID and API key, LDAP and AWS credentials.	
	8	2	Cyberark Conjur can be restarted without any human input.	
	9	2	The documentation covers all necessary information about installation, configuration and usage.	
	10	2	Conjur support the use of an HSM for greater security.	
	11	2	Conjur can be used in a commercial project.	
	12	2	Cyberark Conjur can easily be implemented as an API client.	
	13	1	The installation process only requires creation of configuration files and running commands to install the server.	
	ID	**Pts**	**OPPORTUNITIES**	**THREATS**
EXTERNAL	14	3	No security related issues have been found concerning Conjur.	
	15	3		The open / closed issue ratio is higher than the average.
	16	3		Less commits than the average have been pushed to the repository.
	17	2	A Slack channel for Conjur is available for technical support.	
	18	1		The popularity of Conjur is lower than the average.
	19	1	The source code of Conjur include unit tests.	
	20	1	The documentations mentions possibilities to extend Conjur by, for example, applying open source projects for easier scalability.	
		Score:	(3+3+3+3+3+2+2+2+2+2+2+1)-(3)+(3+2+1+1)-(3+3+1)=25	

Figure 5.1 shows the final score of the compared key management system and Hashicorp Vault has the highest score. Square Keywhiz ends up getting the lowest score from the SWOT analysis.

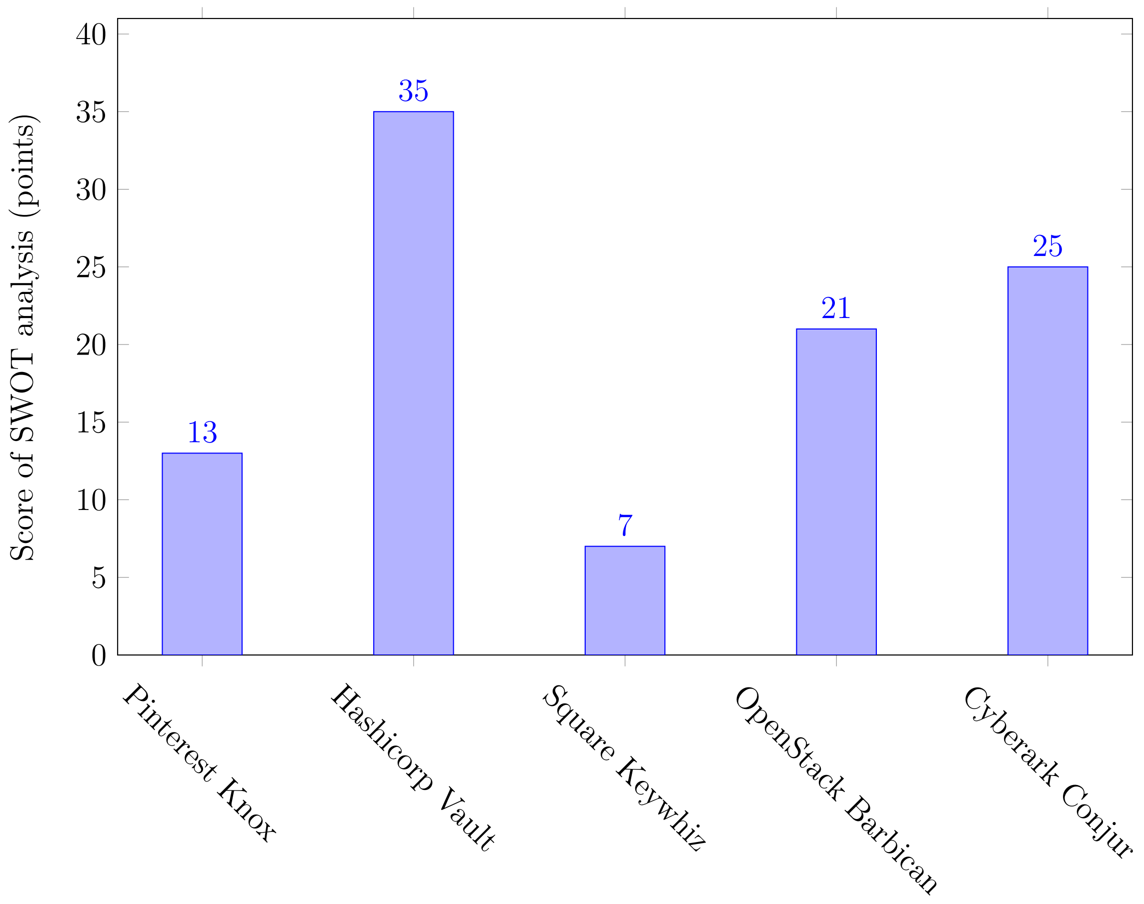

Figure 5.1: A comparison of the final score of the SWOT analysis between the KMS. Higher is better.

5.3 Research question 3

It is important that a KMS is set up properly to get the most out of it. The third research question strives to highlight the best practice of implementing and using Hachicorp Vault, which was concluded as the most suitable for small businesses in this thesis. According to Hashicorp [30], there are some points to consider when implementing Vault in an optimal way.

It is essential that the infrastructure that runs Vault is as secure as possible. Vault should, therefore, be the only main tenancy on the server to prevent vulnerability threats caused by other applications or services. Vault should also be running as a regular non-privileged user, instead of a root user, to minimize the risk of exposing the keys through Vaults process memory by limiting its privileges. It is possible to host Vault on a virtualized environment. However, it is recommended to run Vault on a physical server. The physical server should be restricted with a firewall that only allows for incoming and outgoing communications to Vault and essential services for the machine. Service like SSH or other remote access services are also included in this restriction. Modifications or additional configuration, are done by authorized users, either through physically accessing the machine or through Vaults API. The swap and core dump on the server should also be disabled to make sure that no sensitive is leaked through insecure storage. For the same reason, the shell command history should also be disabled. During initialization, a root token is generated. This token should only be used for the first setup process and revoked after. Vault should also be frequently updated according to instructions from Hachicorp [30].

Some other essential components to a best practice configuration of Vault is the communication and auditing. For communication, TLS should always be used in end-to-end communication to ensure that data in transit is kept secure. When it comes to auditing, it should be enabled to keep a trail of who performs what action at what time [30].

Finally, Vault can be run in, what Hachicorps calls, a high availability mode. The high availability mode is recommended as the way to run Vault by Hachicorp. However, it requires a backend that can support it, but once such a backend is configured the high availability mode is automatically used. To support a high availability mode, Hashicorp has developed its own storage backend called Consul, which is recommended way to run Vault. Vaults high available mode allows multiple Vault servers to utilize a shared backend. The high available mode means that one server, called the active server, runs as normally while other servers are on standby. If requests are made to a standby server, it is automatically redirected and handled on the active server. If the active server is unable to process a request for any reason, a standby server will become the new active server and handle it, and thus achieving high availability [30, 29].

Chapter 6

Analysis and Discussion

This chapter presents an analysis and discussion mainly based on the findings from the chapter about the results, but also a discussion regarding the chosen methodology. The chapter is divided into a section for each research question defined in the thesis.

6.1 Research question 1

Research question 1 regard identified KMSs. An essential part of recommending suitable KMSs for small businesses is to identify and research available options. By identifying available KMSs, they can be further researched to establish how suitable they are for small businesses defined by the business constraints and the security policy. Multiple scientific papers and non-scientific results from Google, i.e., various blogs and forums, were used as sources during the identification process. Despite this, there probably exists KMSs that have been excluded, due to not being identified. However, we believe that it is unlikely that such a solution would be better suited, since if so it would have been more talked about and thus would have been found.

A total of 54 KMSs were found during the research. Out of these, 30 are open source, and 24 are closed source. The open source alternatives range from KMSs that seem to be built for in-house usage and then later published to Github, like Pinterest Knox, to KMSs that are a part of a company's business plan, like Hachicorp Vault. A similar variation of complexity can also be seen in the closed sourced KMSs were some of the KMSs are available as a service, such as Amazon AWS KMS or Microsoft Azure Key Vault. Others are hosted locally, such as Thales Vormetric Data Security Manager.

The thesis was originally planned to feature comparisons between free open sourced KMSs and paid closed sourced KMSs. By comparing KMSs at different costs, the reader can get a better understanding of the differences (if any) that comes with a paid KMS, and thus deciding if it justifies the cost. During the KMs identification process, multiple companies specialized in security and KMSs were contacted. However, due to mostly NDA related problems, no collaboration could be established. Consequently, due to time constraints, we decided to focus on free KMSs based on open source instead of paid KMSs based on closed source. Because of this, there are probably many more paid KMS that are unidentified in this thesis, than free, open source KMSs.

6.2 Research question 2

Research question 2 regards how suitable the identified KMSs is for a small business. A KMSs suitability is mostly decided by the needs and requirements of the businesses that utilize it. Therefore, the first step towards deciding a KMS suitability is to define the business requirements.

6.2.1 Identifying requirements

A KMSs requirement is, in this thesis, defined by the business constraints and the security policy of a fictive business. The business constraints and the security policy intends to give the reader an understanding of the target audience for the comparison specific to this thesis. We believe that our requirements achieve a good compromise between security and usability. However, some points from the security policy might need further discussion.

The chosen level of logging favor traceability over performance and disk usage, to make sure that all actions can be traced back to a user. The communication between clients and the KMS needs to be secure to prevent leakage due to snooping. For usability, it would be preferable if the KMS were to be automatically started on boot since such a KMS would have a less negative impact on its clients during an unplanned reboot. However, it would require that the master key gets saved unencrypted, or at least with information to decrypt it, to the disk. If a malicious user were to gain access to the server, this could potentially compromise the secrets. Consequently, such an approach would require more security around the machine, such as a firewall with strict access rules. It is therefore not required that the KMS can boot automatically, besides if it was necessary, it could probably be scripted regardless.

The security policy also states that the keys and secrets should be on the premises. Today, multiple businesses offer key management as a service. Utilizing a key management service can bring advantages in usability and security, and can be especially appealing for companies that feel uncomfortable with key management. The usability can be improved since key management as a service require less initial configuration. However, the usability can also suffer since there is less control of the system, so the service can potentially be offline due to problems at the service provider. The same goes for security. The security can be improved due to the service providers experience in the field, and the fact that they have the responsibility of backups, so keys are accessible. However, security can also suffer since the service provider can leak or lose access to keys. Therefore, utilizing key management as a service requires trust in the service provider, and it is important to research exactly what responsibility the service provider has.

A KMS's requirements are individual for each business. Different businesses have different views on security and usability, as well as different needs and requirements. Consequently, if a different business were to set up business constraints and a security policy according to their needs instead, the requirements for the KMS would probably be different. It is, therefore, essential that businesses successfully define their own needs to evaluate the suitability of different KMSs.

6.2.2 SWOT analyses

Once the requirements for the KMS are decided, identified KMSs can be evaluated based on the suitability. The evaluation process presented in this thesis first evaluates identified KMSs based on the requirements defined by the business constraints and the security policy. However, to perform a more in-depth evaluation, the KMSs need to be evaluated based on their qualitative attributes as well. The qualitative attributes are evaluated using SWOT analysis, with weighted scores based on the attributes priority. The attributes are derived from the installation, usage, and estimated future support for the KMSs. The priority-based weighted score is used to make sure the attributes that are more critical to the fictive company's business is considered higher. This also makes sure that no attribute alone can affect the choice of which key management system that best fit small business.

Some attributes in the SWOT analysis can be interpreted as strict yes or no questions in most cases. However, sometimes the same attribute can be much more difficult and complex to answer. The attribute regarding whether the KMS has well-written documentation is a clear example of such an attribute. The attribute itself is simple since the documentation is either well written or not. However, it is hard to define good documentation so the answer might be debatable. This attribute was added to the SWOT analysis during the evaluation of the first KMS due to how badly documented it was. The documentation for some KMSs was so inadequate that we were forced to understand parts of the source code to be able to install them. It is also worth mentioning that multiple KMSs lacked the proper instructions to make them ready for a production environment.

Other attributes are based on a comparison, like those regarding the performance or additional features. Optimally, we would like to compare as many identified KMSs as possible. However, due to time constraints, it was only possible to compare the five KMSs evaluated in the SWOT analyses. Consequently, the average value of some attributes is easily affected by outliers in the small sample size of five tools. Some clear examples of outliers are Hachicorp Vaults popularity and Barbicans response time. Despite this, we chose to keep the outliers in the dataset since the overall score of the SWOT analysis would not be significantly affected.

The SWOT analysis was performed on the following KMSs that all seem equally suitable just based on the KMS requirements:

Pinterest Knox

Pinterest Knox ended up with 13 points. Out of the five KMSs evaluated, this represents the fourth best-suited option. The fourth best, or second worst, out of the five evaluated systems seems reasonable based on its quality.

The documentation of Knox was a huge letdown. A brief reading of the documentation makes it seem detailed and well written, with a rigorous background about the tool. However, as an implementation guide, it was hard to follow, and we were forced to understand and edit parts of the source code to successfully implement it. The reason for the lack of quality in the documentation and implementation steps might be due to Knox being built for Pinterest's use case, and not for the general public, which could also explain the low activity on Knox's Github repository. A low activity can be considered bad since there is only a small community interested in keeping Knox in good shape. Over time, this could have a negative impact on the quality since bugs and security vulnerabilities, if any, are less likely to be found and resolved.

One positive attribute identified during the evaluation was the preparations made for Knox to be extended with new storage or authentication backends. Preparing Knox for other backends facilitates the process of replacing it for users who prefer another backend that better suits their needs.

Hachicorp Vault

With a result of 35 points from the SWOT analysis, Hachicorp Vault is concluded to be the most suited KMS for small businesses. The overall impression of Vault was good from the start. With professional documentation and educational guides, Vault was estimated to score high in the evaluation. Vault has a wide range of authentication methods and storage backends to choose from, which makes it suitable for multiple different businesses.

One major reason for Vaults success can be due to its business model. Hachicorp offers Vault based on a freemium business model, which is a business model where the entry model of the version is free and additional tiers costs. This business model means that Vault is made for business purposes, compared to some other evaluated KMSs, which are not. The freemium business model also means that Vault can be extended in the paid version, for example, can HSM support be included.

The only two properties of the SWOT analysis that are in the weakness quadrant does not necessarily have to be considered a weakness. One of these attributes is the amount of data that is sent when requesting secrets. Compared to the other KMSs, Vault sends more data than average because it includes more metadata. In the evaluation present in this thesis, this is regarded negative since the attribute only evaluates the amount of data. The metadata might, however, be useful in practice depending on the application. The other attributes that were considered negative for Vault were the function of sealing/unsealing the backend. In practice, this means that Vault needs input at startup to access the backend, a security feature to prevent unauthorized users from starting Vault. In the evaluation, this is considered negative since Vault cannot start on its own, which negatively impacts the usability of it, but it does increase the security.

Square Keywhiz

At seven points in the SWOT analysis, Square Keywhiz is considered the least suitable of the KMSs evaluated. Similar to Knox, this was expected due to the even worse documentation. The documentation for Keywhix includes numerous broken links and is therefore not complete. Consequently, the source code was used for successfully installing the KMS. Similar to Knox, Keywhiz is probably also built for a specific use case, which can explain the bad maintenance of the documentation.

The high ratio of open issues and the low number of commits suggest that a low amount of work is put into Keywhiz. The low amount of work could explain a security-related issue mention in the Github repository that has not been addressed during the last three years. The nature of the mentioned security issue might suggest that the collaborators of Keywhiz have evaluated it from a security perspective, which suggests an interest in security, which is positive. However, if vulnerabilities are not addressed, this is even worse since it makes the vulnerabilities public.

One positive feature of Keywhiz is its ability to be extended with a client that fetches secrets to the filesystem, a user-friendly way to handle the management since it removes the requirement of implementing REST API calls in the application. Secrets can than be read as a normal file from the filesystem.

OpenStack Barbican

OpenStack Barbican was evaluated to be the third most suitable KMS. At 21 points the overall result of the SWOT analysis was rather good and is a good representation of the reality of using Barbican. Barbican might be favored over other KMSs for businesses that already utilize OpenStack for their infrastructure. A great advantage of Barbican is the ability to extend its security with an HSM using the interoperability standards KMIP and PKCS11, which enables a wide range of available HSMs.

A very noticeable result during the usage of Barbican was the response time, which was much higher than the other key management systems. The reason for this might be the fact that Barbican was implemented together with other modules of the OpenStack suite while the other KMSs was implemented as a standalone. An authentication request to Barbican had to be rerouted in OpenStack to the Keystone component and then back to Barbican which probably adds some latency. However, there are ways to avoid the implications that latency brings. For example, the application that utilizes Barbican could be designed to periodically retrieve and store secrets in volatile memory for fast access, which sacrifices some security for usability.

Cyberark Conjur

At 25 points, Cyberark Conjur is concluded to be the second most suitable KMS out of the five evaluated in SWOT analyses. The biggest advantage of Conjur is its simple installation process using Docker. By utilizing the docker image, a business can be up and running with a default version of Conjur within a few minutes. Cyberark also provides the possibility to try Conjur through their hosted software without having to install anything, which makes it a good alternative to test the concept of KMSs.

The documentation of the installation and configuration process of Conjur was well written and easy to follow. However, one disadvantage, from our perspective of Conjur, was the fact that it did not support certificates as a way of authenticating. There are, however, other possible ways to authenticate to Conjur, which could be suitable for small businesses.

6.3 Research Question 3

The third and last research question regards the best practice method of implementing the most suitable KMS, Hachicorp Vault. Once a KMS has been implemented in the infrastructure, it is an essential part of it, and it is therefore also essential that the KMS is configured according to best practice.

Based on the research performed, Vault should be configured with Hachicorps own, high availability, backend Consul, which automatically makes Vault run in a high availability mode. Both Vault and Consul should be installed as the only main service on restricted machines to minimize the security threats caused by other applications and services on the system. Restricted machines imply that the machines are restricted by a firewall to only allow communication to and from Vault, and services, essential to the system. Apart from the system and the infrastructure as a whole, Vault should also be configured to communicate over TLS and audit every action the system does.

It is worth noting that it could be argued that a key management system does not entirely solve the issue of handling software secrets in software projects. Since an application needs some method of authenticating against the KMS, there is still some information that must be kept secure on the applications side, whether it is a certificate, API key, or credentials, like username and password. All of these, have to be accessible from the application, either as data in the source code or as a resource that is loaded at runtime, like a certificate or a configuration file. As mentioned in the introduction, secrets should not be included in the source code. However, the alternative of an external resource that is read during runtime is not necessarily more secure in every aspect since it too can be exploited if leaked. Consequently, it is essential that data, used for authentication against a KMS, is secure at rest, and shared between the users who need it securely.

To conclude the analysis and discussion, a KMSs advantage is more noticeable when it comes to the management of secrets rather than the security of them. Although it brings some advantages to security since the secrets can be easily rotated or managed in general, it also comes with some disadvantages since secrets are collected in the same backend service. Depending on the KMS implemented and the configuration, a KMS can also be a single point of failure. All of these aspects are important to keep in mind when the potential of implementing a KMS is discussed.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

There is currently a huge selection of key management systems available for companies to handle their encryption keys and software secrets. The quality, price, and intended use all vary, thus making it hard for businesses to decide on a system to implement, especially small businesses with more limited resources than their larger counterparts. The thesis aims to facilitate this process by identifying and evaluating available key managers.

The thesis identified 54 key management systems. Out of these systems, 30 are open source systems available at no cost, and 24 are paid close sourced systems. For evaluating key management systems suitability for small businesses, the thesis defines fictive requirements of a key management system, based on its fictive business constraints and security policy. The fictive business is defined in collaboration with Miljödata to match other small businesses with similar needs. The requirements for a key management system are used to filter the 54 systems. This filtering results in five systems that meet the requirements. The five systems are extensively evaluated using SWOT analyses to determine the order of how suitable they are for small businesses. The SWOT analyses resulted in the following order of suitability of the five systems with the most suited first: Hashicorp Vault, Cyberark Conjur, OpenStack Barbican, Pinterest Knox, and lastly Square Keywhiz.

The evaluation concluded that Hachicorp Vault is the most suited key management system with a total of 35 points. For best practice, Hachicorp Vault should be configured and installed with Hachicorps own storage backend, Consul, on physical machines with restricted access. Hachicorp Vault should only communicate securely using TLS, and the machines, Hachicorp Vault, and other running services should be actively updated to mitigate unauthorized access.

The entire content of this thesis can be used by small businesses, at least, as a guideline of how to perform an evaluation themselves. The results of this thesis can also be used either partly or fully by a small company if their requirements of a KMS are partly or completely equal to the requirements outlined in this thesis.

7.2 Future work

The thesis only scratches the surface of key management. Although 54 key management systems are identified, only five are installed and extensively evaluated.

There are multiple ways to extend the works done in this thesis in future work. One example of this is to perform the same extensively evaluation on more systems. Evaluating more systems would give a broader perspective of the use cases and limitations from various KMSs. It would also provide a better comparison for those attributes from the SWOT analysis that directly compare the KMSs against each other.

Another example could be to extend the evaluation method. The evaluation method presented in this thesis evaluates the installation process and parts of how it is to use the KMS. The evaluation method could, therefore, be extended to feature attributes essential for long time use, like the update process or how easy it is to backup and restore the systems. These were initially planned to be featured in the thesis but was later discarded due to time constraints. The evaluation process could also be extended with a vulnerability assessment and penetration testing to make it more security oriented.

Finally, another aspect that was initially planned for in this thesis was a comparison between paid and free KMSs. Such a comparison could evaluate whether the advantages, if any, with a paid KMS, are worth the extra cost. The thesis mentions some advantages with key management as a service in the discussion section. However, it would be beneficial to research whether paid KMSs, that are installed locally, brings any advantages compared to Hashicorp Vault.

References

[1] Authentication. https://docs.conjur.org/latest/en/Content/Operations/Services/Authentication-new.htm?tocpath=Fundamentals%7CAuthentication%7C_____0. (Accessed on 06/14/2019).

[2] Credential security with secrets management, rbac and machine identity. https://www.conjur.org/why-conjur/concepts/. (Accessed on 06/15/2019).

[3] Github - cyberark/conjur: Cyberark conjur automatically secures secrets used by privileged users and machine identities. https://github.com/cyberark/conjur. (Accessed on 06/15/2019).

[4] Github - cyberark/conjur-quickstart: Start securing your secrets and infrastructure by installing conjur, using docker and the official conjur containers on dockerhub. https://github.com/cyberark/conjur-quickstart. (Accessed on 06/15/2019).

[5] Github - cyberark/conjur-tutorials: A repository for tutorials related to conjur. https://github.com/cyberark/conjur-tutorials. (Accessed on 06/15/2019).

[6] Github - openstack/barbican: Barbican is a rest api designed for the secure storage, provisioning and management of secrets, including in openstack environments. https://github.com/openstack/barbican. (Accessed on 06/14/2019).

[7] Github - openstack/python-barbicanclient: Client library for barbican api. https://github.com/openstack/python-barbicanclient. (Accessed on 06/14/2019).

[8] Hashicorp vault: Enterprise pricing, packages & features. https://www.hashicorp.com/products/vault/enterprise. (Accessed on 06/13/2019).

[9] Host. https://docs.conjur.org/Latest/en/Content/Operations/Policy/statement-ref-host.htm?TocPath=Fundamentals%7CPolicy%20Management%7CPolicy%20Statement%20Reference%7C_____5. (Accessed on 06/14/2019).

[10] Openstack docs: Install and configure for ubuntu. https://docs.openstack.org/barbican/latest/install/install-ubuntu.html. (Accessed on 06/14/2019).

[11] Security policy as code. `https://docs.conjur.org/Latest/en/Content/Operations/Policy/policy-intro.html?tocpath=Fundamentals%7CPolicy%20Management%7CSecurity%20policy%20as%20code%7C_____0`. (Accessed on 06/15/2019).

[12] User. `https://docs.conjur.org/Latest/en/Content/Operations/Policy/statement-ref-user.htm?TocPath=Fundamentals%7CPolicy%20Management%7CPolicy%20Statement%20Reference%7C_____11`. (Accessed on 06/14/2019).

[13] Vault documentation - vault by hashicorp. `https://www.vaultproject.io/docs/`. (Accessed on 03/21/2019).

[14] vault/license at master · hashicorp/vault · github. `https://github.com/hashicorp/vault/blob/master/LICENSE`. (Accessed on 06/13/2019).

[15] Regulation (ec) no 679/2016 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). *OJ*, L 119:51–52, 2016-05-04.

[16] Bushra AlBelooshi, Ernesto Damiani, Khaled Salah, and Thomas Martin. Securing cryptographic keys in the cloud: A survey. *IEEE Cloud Computing*, 3(4):42–56, 2016.

[17] S. A. Aljawarneh, A. Alawneh, and R. Jaradat. Cloud security engineering: Early stages of SDLC. *Future Generation Computer Systems*, 74:385–392, 2017.

[18] BenchmarkDotNet. Frequently asked questions. `https://benchmarkdotnet.org/articles/faq.html`. [Online; accessed 2019-05-08].

[19] Mathias Björkqvist, Christian Cachin, Robert Haas, Xiao-Yu Hu, Anil Kurmus, René Pawlitzek, and Marko Vukolić. Design and implementation of a key-lifecycle management system. In *International Conference on Financial Cryptography and Data Security*, pages 160–174. Springer, 2010.

[20] A. R. Buchade and R. Ingle. Key management for cloud data storage: Methods and comparisons. In *Advanced Computing & Communication Technologies (ACCT), 2014 Fourth International Conference on*, pages 263–270. IEEE, 2014.

[21] L. Chadwick. My run in with Unauthorised Litecoin mining on AWS. `http://vertis.io/2013/12/16/unauthorised-litecoin-mining.html`, 2013. [Online; accessed 2019-01-07].

[22] Ramaswamy Chandramouli, Michaela Iorga, and Santosh Chokhani. Cryptographic key management issues and challenges in cloud services. In *Secure Cloud Computing*, pages 1–30. Springer, 2014.

[23] Jolyon Clulow. On the security of pkcs# 11. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425. Springer, 2003.

[24] Cryptomathic. Selecting the right key management system. https://www.cryptomathic.com/hubfs/Documents/White_Papers/Cryptomathic_White_Paper_-_Selecting_The_Right_Key_Management_System.pdf, 2019. [Online; accessed 2019-03-19].

[25] Cyberark. Get Started. https://www.conjur.org/get-started/. [Online; accessed 2019-04-15].

[26] Ed Dawson, Andrew Clark, and Mark Looi. Key management in a non-trusted distributed environment. *Future Generation Computer Systems*, 16(4):319 – 329, 2000.

[27] Rion Dooley, Andy Edmonds, David Y Hancock, John Michael Lowe, Edwin Skidmore, Andrew K Adams, Ryan Kiser, Mark Krenz, Von Welch, and Richard Knepper. Security best practices for academic cloud service providers. Technical report, 2018.

[28] Anders Fongen. Identity management and integrity protection in the internet of things. In *2012 third international conference on emerging security technologies*, pages 111–114. IEEE, 2012.

[29] Hashicorp. High Availability. https://www.vaultproject.io/docs/internals/high-availability.html. [Online; accessed 2019-05-16].

[30] Hashicorp. Production hardening. https://learn.hashicorp.com/vault/operations/production-hardening. [Online; accessed 2019-05-16].

[31] Michael Kretzschmar and Sebastian Hanigk. Security management interoperability challenges for collaborative clouds. In *Systems and Virtualization Management (SVM), 2010 4th International DMTF Academic Alliance Workshop on*, pages 43–49. IEEE, 2010.

[32] Maxvt. Infrastructure Secret Management Software Overview. https://gist.github.com/maxvt/bb49a6c7243163b8120625fc8ae3f3cd, 2016. [Online; accessed 2019-02-28].

[33] M. McPherrin. Distrust the Database 182. https://github.com/square/keywhiz/issues/182. [Online; accessed 2019-04-05].

[34] Asim Mehmood. Hsms and key management: Effective key security. https://www.cryptomathic.com/news-events/blog/hsms-and-key-management-effective-key-security, Jan 2018. [[Online; accessed 2019-03-19].

[35] Ponemon Insitute nCipher Security. Global Encryption Trends Study. https://www.ncipher.com/2019/global-encryption-trends-study, 2019. [Online; accessed 2019-05-10].

[36] OASIS. KMIP Implementations known to the KMIP TC. https://wiki.oasis-open.org/kmip/KnownKMIPImplementations. [Online; accessed 2019-02-28].

[37] OASIS. OASIS Wiki List. https://wiki.oasis-open.org/. [Online; accessed 2019-05-13].

[38] Openstack. Barbican Documentation. https://docs.openstack.org/barbican/latest/. [Online; accessed 2019-03-31].

[39] Fabien AP Petitcolas. Kerckhoffs' principle. *Encyclopedia of cryptography and security*, pages 675–675, 2011.

[40] Pinterest. Knox Wiki. https://github.com/pinterest/knox/wiki/, 2016. [Online; accessed 2019-03-14].

[41] Pinterest. Similar Solutions. https://github.com/pinterest/knox/wiki/Similar-Solutions, 2016. [Online; accessed 2019-02-28].

[42] J.H. Reinholm. Key Management: Explaining the Life Cycles of a Cryptographic Key. https://www.cryptomathic.com/news-events/blog/key-management-the-life-cycles-of-a-cryptographic-key. [Online; accessed 2019-05-13].

[43] Townsend Security. The definitive guide to encryption key management fundamentals. https://info.townsendsecurity.com/definitive-guide-to-encryption-key-management-fundamentals. [Online; accessed 2019-05-13].

[44] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani. Detecting and mitigating secret-key leaks in source code repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 396–400. IEEE, 2015.

[45] D. Sitaram, S. Harwalkar, U. Simha, S. Iyer, and S. Jha. Standards based integration of advanced key management capabilities with openstack. In *Cloud Computing in Emerging Markets (CCEM), 2015 IEEE International Conference on*, pages 98–103. IEEE, 2015.

[46] Square. Keywhiz. https://github.com/square/keywhiz. [Online; accessed 2019-04-05].

[47] Square. Keywhiz. https://square.github.io/keywhiz/. [Online; accessed 2019-04-05].

[48] Square. Keywhiz Wiki. https://github.com/square/keywhiz/wiki. [Online; accessed 2019-04-05].

[49] Stackshare. Secrets Management. https://stackshare.io/secrets-management. [Online; accessed 2019-02-28].

[50] G Stigestadh and F Moberg. Big data, på gott och ont: Gdpr som stöd för den enskilde individen. http://lup.lub.lu.se/student-papers/record/8947988, 2018. [Online; accessed 2019-02-06].

[51] Jon Topper. Compliance is not security. *Computer Fraud & Security*, 2018(3):5–8, 2018.

[52] Wikipedia. Key management. `https://en.wikipedia.org/wiki/Key_management`. [Online; accessed 2019-02-28].

[53] C. Wohlin. A snowballing procedure for systematic literature studies and a replication. *Proceedings 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, pages 321–330, 2014.

Appendix A

Documentation of Pinterest Knox

A.1 Prerequisites

- Linux Ubuntu 18.04
- Go 1.12
- MySQL 14.14

A.2 Implementation process

Pinterest Knox was downloaded and compiled using Go. Knox could be installed and used based on the instructions presents on its Github page. However, this used default settings with server and client certificates, which are also published in the same Github repository. The Knox server was, therefore, customized during installation to be ready for production. The customization to the server was done in a file called *main.go*, located in the *dev_server* subfolder. To be able to set up the Knox server and connect a client to it, the following three certificates had to be generated. A client certificate used for authenticating devices, a server certificate to secure communication to the server, and a root certificate to sign the server and client certificate.

By default, Knox uses a temporary database stored in memory. However, the data stored is lost if the server dies, thus it should not be used in production. Consequently, the Knox wiki contains instructions to replace the temporary database with a backend consisting of either a MySQL database or a PostgreSQL database. The implementation present in this thesis uses a MySQL backend. However, a PostgreSQL backend would probably have worked just as well. Apart from the backend database, the master key that encrypts the secrets should also be replaced for production since it is otherwise known. The master key was easily changed in the *main.go* file located in the *dev_server* subfolder, by replacing the existing key. For this installation, a new key was generated using OpenSSL and saved to a file, which is read by the Knox server at startup. After these modifications, the server was installed and started as a background process.

Keys are added to Knox through a CLI client. This client could also have been run as a daemon that cache keys locally and keeps them up to date. The daemon frequently communicates with the Knox server to see if keys have been rotated or deactivated/activated to ensure that the right keys are always cached. By locally caching keys, they can be read from the local filesystem instead which makes them available even if the Knox server goes down. The developers for Knox recommends that this client should be installed and run on every machine that accesses keys from the Knox server. However, during this thesis, this client was only used for adding keys to Knox. When keys are added, the user can specify what permission other groups, users or machines should have for it. For this thesis, the keys were added with reading permission the machine with the client certificate previously generated. When adding keys to Knox, the user must be authenticated through a Github Access Token, which is also the only way to authenticate users for Knox. The other type of authentication, with certificates, are called machine authentication and can only be used to read keys, not modify.

The test project communicates with the Knox API through a wrapper class that can initiate and authenticate the connection as well as accessing the desired keys. This class was constructed to be a part of the factory, which can easily create it for unit testing and benchmarking.

A.3 Implementation steps

1. A root, client and server certificate is needed to securely authenticate and communicate with the Knox server. These can either be signed by a Certificate Authority or be self signed. In this implementation self signed certificates will be used and can be generated as following:

```
$ openssl req -new -x509 -days 365 -keyout ca-key.pem -out ca-crt.pem

$ openssl genrsa -out server-key.pem 4096
$ openssl req -new -sha256 -key server-key.pem -out server-csr.pem
$ openssl x509 -req -days 365 -in server-csr.pem -CA ca-crt.pem \
    -CAkey ca-key.pem -CAcreateserial -out server-crt.pem

$ openssl genrsa -out client-key.pem 4096
$ openssl req -new -sha256 -key client-key.pem -out client-csr.pem
$ openssl x509 -req -days 365 -in client-csr.pem -CA ca-crt.pem \
    -CAkey ca-key.pem -CAcreateserial -out client-crt.pem
```

2. A master encryption key have to be generated to be used for encrypting the keys. It's cruzial this key is kept safe, in this implementation it is created on the server and kept as a file on that machine. It can be further secured by adding it to an external HSM or another KMS.

```
$ openssl rand -hex -out master-enc-key.dat 32
```

3. Export the generated client certificate to a .pfx file and install it on each machine that will communicate with Knox.

```
$ openssl pkcs12 -export -out client.pfx -inkey client-key.pem \
    -in client-crt.pem -certfile ca-crt.pem
```

4. Download the Knox repository using Go.

```
$ go get -d github.com/pinterest/knox
```

5. Download the dependency for communicating with SQL.

```
$ go get -u github.com/go-sql-driver/mysql
```

6. Edit the installation file *$GOPATH/src/github.com/pinterest/knox/cmd/dev_server/main.go* for Knox server and remove the following lines for handling certificates:

```
const caCert = `...`

...

tlsCert, tlsKey, err := buildCert()
if err != nil {
        errLogger.Fatal("Failed␣to␣make␣TLS␣key␣or␣cert:␣", err)
}
```

Replace it with this code snippet:

```
tlsCert, err := ioutil.ReadFile("/path/to/server-crt.pem")
if err != nil {
        errLogger.Fatal("Failed␣to␣get␣tlsCert:␣", err)
}

tlsKey, err := ioutil.ReadFile("/path/to/server-key.pem")
if err != nil {
        errLogger.Fatal("Failed␣to␣get␣tlsKey:␣", err)
}

caCert, err := ioutil.ReadFile("/path/to/ca-crt.pem")
if err != nil {
        errLogger.Fatal("Failed␣to␣get␣caCert:␣", err)
}
```

7. Edit the installation file *$GOPATH/src/github.com/pinterest/knox/cmd/dev_server/main.go* for Knox server and remove the following line for handling the database:

```
db := keydb.NewTempDB()
```

Replace it with the following code snippet:

```
d, err := sql.Open("mysql", "user:password@host/knoxdb")
if err != nil {
        //handle the error
}
db, err := keydb.NewSQLDB(d)
```

And add the following to the *import* section:

```
"io/ioutil"
"database/sql"
_ "github.com/go-sql-driver/mysql"
```

8. Edit the installation file *$GOPATH/src/github.com/pinterest/knox/cmd/dev_server/main.go* for Knox server and remove the following line with the default encryption key:

```
dbEncryptionKey := []byte("testtesttesttest")
```

Replace it with the following code snippet:

```
dbEncryptionKey, err := ioutil.ReadFile("/path/to/master-enc-key.dat")
if err != nil {
        errLogger.Fatal("Failed␣to␣get␣encryption␣key:␣", err)
}
```

9. Install the Knox server.

```
go install github.com/pinterest/knox/cmd/dev_server
```

10. Start the Knox server as a background service.

```
$GOPATH/bin/dev_server keymanager.fictive.ab &
```

11. Add a secret that should be kept secure in Knox and change the access policy for the secret.

```
$ curl -k -H "Authorization:␣0u<Github␣Access␣Token>"  \
    -H "Content-Type:␣application/x-www-form-urlencoded"  \
    -d "id=<Secret␣ID>&data=<Base64␣encoded␣secret>"  \
    -X POST https://keymanager.fictive.ab:9000/v0/keys/

$ curl -k -H "Authorization:␣0u<Github␣Access␣Token>" \
    -H "Content-Type:␣application/x-www-form-urlencoded" \
    -d "access=<Base64␣encoded␣JSON␣object␣with␣ACL␣info>" \
    -X PUT https://keymanager.fictive.ab:9000/v0/keys/APIKey/access/
```

Appendix B
Documentation of Hashicorp Vault

B.1 Prerequisites

- Linux Ubuntu 18.04
- MySQL 14.14
- Unzip

B.2 Implementation process

Hachicorp Vault can be downloaded as a pre-built binary or build from source. The installation covered in this thesis uses the pre-built version due to convenience. The binary could then be unpacked and placed into the */bin* folder, according to Hachicorps instructions. For local and experimental purposes, Hachicorp provides, what they call Vault dev server. The dev server is a built-in, pre-configured server that uses an in-memory database. This server should not be used in production, but it is a good way to test some of the advantages with a KMS. For production use, Vault needs to be configured through an HCL (Hashicorp Configuration Language) configuration file. The HCL files specify settings for the listener and the storage. For the installation process documented in this thesis, Vault was configured to communicate over TLS with clients and use a MySQL database backend for storage. Vault could then be started, and the environment was initiated. The initiation happens every time Vault is started against a new backend and is needed to generate the necessary keys. After a successful initiation process, Vault outputs a root token used for authenticating the root user at login, and five unseal keys. Using default settings, three of these keys are needed at startup for Vault to decrypt data in the backend storage. Thus, the keys should be distributed to different people or stored at different locations to mitigate the damage one rogue user could potentially inflict.

Once Vault has been initiated, the root user could log in using the previously mention root token. Vault was then configured to allow clients to authenticate through certificates, together with an access-policy that permits users to only read secrets from a specific path. Lastly, secrets were added from the terminal. However, secrets can also be added using a web request.

After Vault was installed, started and configured with the needed secrets, curl was used to verify that secrets could be successfully retrieved. A wrapper class, similar to the one for Knox, was then implemented to the factory. similar to Knox, this wrapper class was then used for unit testing and benchmarking.

B.3 Implementation steps

1. A root, client and server certificate is needed to securely authenticate and communicate with the Knox server. These can either be signed by a Certificate Authority or be self signed. In this implementation self signed certificates will be used and can be generated as following:

```
$ openssl req -new -x509 -days 365 -keyout ca-key.pem -out ca-crt.pem

$ openssl genrsa -out server-key.pem 4096
$ openssl req -new -sha256 -key server-key.pem -out server-csr.pem
$ openssl x509 -req -days 365 -in server-csr.pem -CA ca-crt.pem \
    -CAkey ca-key.pem -CAcreateserial -out server-crt.pem

$ openssl genrsa -out client-key.pem 4096
$ openssl req -new -sha256 -key client-key.pem -out client-csr.pem
$ openssl x509 -req -days 365 -in client-csr.pem -CA ca-crt.pem \
    -CAkey ca-key.pem -CAcreateserial -out client-crt.pem
```

2. Export the generated client certificate to a .pfx file and install it on each machine that will communicate with Vault.

```
$ openssl pkcs12 -export -out client.pfx -inkey client-key.pem \
    -in client-crt.pem -certfile ca-crt.pem
```

3. Download the latest version available for Hashicorp Vault, in this implementation Vault 1.0.3 is used.

```
$ wget https://releases.hashicorp.com/vault/1.0.3/vault_1.0.3_linux_amd64.zip
```

4. Unzip the downloaded file and move the binary to */bin*

```
$ unzip vault_1.0.3_linux_amd64.zip -d /bin
```

5. Create and edit a configuration file *vault-config.hcl* and paste the following configuration to the configuration file:

```
listener "tcp" {
  address     = "192.168.10.127:8200"
  tls_disable = "false"
  tls_cert_file = "/path/to/server-crt.pem"
  tls_key_file = "/path/to/server-key.pem"
  tls_require_and_verify_client_cert = "true"
  tls_client_ca_file = "/path/to/ca-crt.pem"
}

storage "mysql" {
  username = "root"
  password = "root"
  database = "vault"
}
```

6. Start the server using the configuration file.

   ```
   $ vault server -config vault-config.hcl
   ```

7. Initialize Vault server environment and save the root token and each unseal key separately on a secure location, such as a physical paper in a physical vault.

   ```
   $ vault operator init
   ```

8. Login as root and enable certificate as the authentication method.

   ```
   $ vault login <initial root_token>
   $ vault auth enable cert
   ```

9. Create and edit a policy file *vault-policy-prod.hcl* and paste the following policy to the policy file:

   ```
   path "secret/*" {
     capabilities = ["read"]
   }
   ```

10. Add the generate client certificate and the defined policy to allow access to the keys.

    ```
    $ vault policy write prod vault-policy-prod.hcl
    $ vault write auth/cert/certs/prod \
        display_name=prod \
        policies=prod \
        certificate=@client-crt.pem \
        ttl=3600
    ```

11. Add a secret that should be kept secure in Vault.

    ```
    curl -k \
        --header "X-Vault-Token:␣<Root␣Access␣Token>" \
        --request POST \
        --data '{"key": "<Secret>"}' \
        https://keymanager.fictive.ab:8200/v1/secret/<Secret ID>
    ```

Appendix C
Documentation of Square Keywhiz

C.1 Prerequisites

- Linux Ubuntu 18.04
- Java Development Kit 11.0.2
- Apache Maven 3.5.2
- MySQL 14.14

C.2 Implementation process

Keywhiz is a Java-based key management system, by Square, that is built using Apache Maven, a build automation tool primarily used for Java projects. Consequently, both the Java Development Kit and Maven was installed on the Ubuntu machine before the Keywhiz installation process could begin. Keywhiz was then installed using the instructions on its webpage, with MySQL for the database backend instead of H2. During the installation process, multiple attempts were made to change the certificates, instead of using those included in the Github repository. However, due to the lack of documentation for making Keywhiz production ready, these attempts were unsuccessful. The installation was therefore continued using the included certificate. A database was then created and migrated according to the installation instructions, and a server administrator was added for the KMS. The server could then be started.

Managing the server, i.e., managing users and keys, are done through a CLI client. This client was as well as the server, built using Maven according to Squares instructions. A test group and test user was created through the CLI client. The test user was supposed to be created to allow authentication through certificates since this is the preferred method for our test application. However, due to lacking documentation, we were unable to specify this when the user was created. The database was therefore manually modified to allow this by changing the value in a column called *automationallowed*, located in the *clients* table. Lastly, the keys were added to Keyqhiz through the CLI client.

After the KMS was installed and configured with a user and keys, Curl was used for verifying that it was possible to retrieve secrets using certificates. After verification, a wrapper class was implemented to the factory in the test application, and later used for unit tests and benchmarking.

C.3 Implementation steps

1. Download the repository for Keywhiz and switch into that directory.

```
$ git clone https://github.com/square/keywhiz.git && cd keywhiz
```

2. Build the server from the source code and choose to use MySQL as the desired storage backend.

```
$ mvn package -X -am -pl server -P mysql
```

3. Generate base derivation key for the encryption process of secrets and cookies.

```
$ java -jar server/target/keywhiz-server-*-SNAPSHOT-shaded.jar gen-aes

$ head -c 32 /dev/urandom | base64 > \
    server/src/main/resources/dev_and_test_cookiekey.base64
```

4. Create the database *keywhizdb_ development*, preview the migration changes and then migrate the database with the latest changes.

```
$ mysql

mysql> CREATE DATABASE keywhizdb_development;
mysql> exit;

$ java -jar server/target/keywhiz-server-*-SNAPSHOT-shaded.jar \
    preview-migrate server/src/main/resources/keywhiz-development.yaml

$ java -jar server/target/keywhiz-server-*-SNAPSHOT-shaded.jar \
    migrate server/src/main/resources/keywhiz-development.yaml
```

5. Add an administrator account to the server.

```
$ java -jar server/target/keywhiz-server-*-SNAPSHOT-shaded.jar \
    add-user server/src/main/resources/keywhiz-development.yaml
```

6. Start the server with the provided configuration file.

```
$ java -jar server/target/keywhiz-server-*-SNAPSHOT-shaded.jar \
    server server/src/main/resources/keywhiz-development.yaml
```

7. Build the CLI from the source code.

```
$ mvn package -X -am -pl cli
```

8. Create an alias for the CLI application and login using the administrator account.

```
$ alias keywhiz.cli="/path/to/keywhiz-cli-*-SNAPSHOT-shaded.jar"

$ keywhiz.cli --devTrustStore --user admin login
```

9. Create a new group and then a new client that is assigned to the created group. Enable the client to authenticate using the certificate through the database.

```
$ keywhiz.cli --devTrustStore --user admin add group --name groupname
$ keywhiz.cli --devTrustStore --user admin add client --name client \
    --group groupname

$ mysql
mysql> USE keywhizdb_development;
mysql> UPDATE clients SET automationallowed = 1 WHERE id = 1;
mysql> exit;
```

10. Add a secret that should be kept secure in Keywhiz.

```
$ echo -n "<Secret>" | keywhiz.cli --devTrustStore --user admin \
    add secret --name <Secret ID> --content true \
    --json {"owner":"client","group":"groupname","mode":"0400"}
```

Appendix D

Documentation of OpenStack Barbican

D.1 Prerequisites

- Linux Ubuntu 18.04
- MySQL 14.14
- OpenStack (Devstack)

D.2 Implementation process

Barbican is the key manager component of OpenStack, an open-source software platform for cloud computing. By searching on the internet, it looks like Barbican can be installed as a stand-alone service. However, to ensure the most realistic results a decision was made to install OpenStack with all of its components instead of a stand-alone version of Barbican. Barbican, with the rest of OpenStack, was installed through DevStack, a series of scripts used to bring up an OpenStack environment based on the latest version of every component. Worth mentioning is that DevStack is not intended as an OpenStack installer, however, it did seem good enough for this use case since the majority of OpenStacks components is irrelevant for an evaluation of Barbican.

After the OpenStack installation was completed, Barbican could be configured using instructions provided by OpenStack[1]. First, a MySQL database was created and granted all privileges. Some admin credentials were then sourced to the environment to gain access to administrator restricted CLI commands. A new user was added with *admin* and *creator* roles, as well as three API endpoints for *public*, *internal* and *admin*, respectively. The installation was then continued by installing *barbican-api*, *barbican-keystone-listener* and *barbican-worker* through the package manager APT. After the packages were installed some changes was made to a configuration file to configure database access, message queue access, and identity service access. Lastly, the database was populated with the correct tables and constraints by running a script, followed by a restart of Barbican and some dependencies.

[1]https://docs.openstack.org/barbican/latest/install/install-ubuntu.html

During installation, several attempts were made to configure mutual TLS as an authentication method for clients as well as between Barbican and Keystone, Barbicans identity service. However, these attempts were unsuccessful and thus a decision was made to use a username and password for authentication instead.

When Barbican was configured and was up and running, keys were added using the CLI, and the access control list was updated to permit the test user to access these keys. Similar to the other evaluations, the API was first tested using curl before a wrapper class was implemented to the factory in the test project. The wrapper class was then used in the unit tests and benchmarking.

D.3 Implementation steps

1. Create a database and a database user in MySQL

```
$ mysql
mysql> CREATE DATABASE barbican;
mysql> GRANT ALL PRIVILEGES ON barbican.* TO 'barbican'@'localhost' \
  IDENTIFIED BY 'BARBICAN_DBPASS';
mysql> GRANT ALL PRIVILEGES ON barbican.* TO 'barbican'@'%' \
  IDENTIFIED BY 'BARBICAN_DBPASS';
mysql> exit;
```

2. Create a file *admin-openrc.sh* containing the following content

```
#!/usr/bin/env bash
export OS_AUTH_URL=http://192.168.10.127/identity/v3
export OS_PROJECT_ID=04dbb94342cd4ff58c362cf40d0cb271
export OS_PROJECT_NAME="demo"
export OS_USER_DOMAIN_NAME="Default"
if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi
export OS_PROJECT_DOMAIN_ID="default"
if [ -z "$OS_PROJECT_DOMAIN_ID" ]; then unset OS_PROJECT_DOMAIN_ID; fi
unset OS_TENANT_ID
unset OS_TENANT_NAME
export OS_USERNAME="admin"
echo "Please␣enter␣your␣OpenStack␣Password␣for␣project␣$OS_PROJECT_NAME
as␣user␣$OS_USERNAME:␣"
read -sr OS_PASSWORD_INPUT
export OS_PASSWORD=$OS_PASSWORD_INPUT
export OS_REGION_NAME="RegionOne"
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
```

3. Source the admin credentials to the OpenStack CLI client.

```
$ source admin-openrc.sh
```

4. Create a user to be used with Barbican.

```
$ openstack user create --domain default --password-prompt barbican
$ openstack role add --project service --user barbican admin
$ openstack role create creator
$ openstack role add --project service --user barbican creator
$ openstack service create --name barbican \
    --description "Key␣Manager" key-manager
```

5. Create key manager API endpoints for Barbican.

```
$ openstack endpoint create --region RegionOne \
  key-manager public http://192.168.10.127:9311
$ openstack endpoint create --region RegionOne \
  key-manager internal http://192.168.10.127:9311
$ openstack endpoint create --region RegionOne \
  key-manager admin http://192.168.10.127:9311
```

6. Update the package tool and install Barbican.

```
$ apt-get update
$ apt-get install barbican-api barbican-keystone-listener barbican-worker
```

7. Edit the file */etc/barbican/barbican.conf* and add the following to the sections *[DEFAULT]* and *[keystone_ authtoken]*.

```
[DEFAULT]
...
sql_connection = \
    mysql+pymysql://barbican:BARBICAN_DBPASS@192.168.10.127/barbican
transport_url = rabbit://openstack:RABBIT_PASS@192.168.10.127

...

[keystone_authtoken]
...
www_authenticate_uri = http://192.168.10.127:5000
auth_url = http://192.168.10.127:5000
memcached_servers = 192.168.10.127:11211
auth_type = password
auth_plugin = password
project_domain_name = default
user_domain_name = default
project_name = service
username = barbican
password = BARBICAN_PASS
```

8. Populate the database with appropriate tables.

```
$ su -s /bin/sh -c "barbican-manage␣db␣upgrade" barbican
```

9. Restart services to apply the changes and enable Barbican

```
$ service barbican-keystone-listener restart
$ service barbican-worker restart
$ service apache2 restart
```

10. Add a secret that should be kept secure in Barbican and update the access control list.

```
$ openstack secret store --name <Secret ID> --payload <Secret>

$ openstack acl user add --user <Client ID> \
    http://localhost:9311/v1/secrets/<Returned ID from last command>/
```

Appendix E

Documentation of Cyberark Conjur

E.1 Prerequisites

- Linux Ubuntu 18.04
- Docker 18.06

E.2 Implementation process

Cyberark Conjur was installed through Docker, due to it being the only available method with instructions from Cyberark. Docker allows for a very straight forward installation process since the dependencies for Conjur are included in the Docker container. After Docker was installed, Conjur was downloaded by cloning its tutorial repository at Github. The tutorial repository was chosen because it is used in a tutorial about getting Conjur production ready with TLS. Besides, during installation, it installs the latest version of Conjur regardless so there should be no difference apart from the TLS utilization after installation.

Conjur was installed by running a shell script. The shell script generated a self-signed certificate using Openssl, an encryption key, creates a Conjur user and starts Conjur as that user. The encryption key and the created users password was saved to files. These files should, therefore, be either stored securely or deleted if another method of storing the API key is preferred. Such a method could, for example, be writing it to a paper and storing it in a physical safe. Conjur was then configured with a security policy.

Adding secrets to Conjur was done by creating and loading variables into Conjur. Variables were created from YML-files that specified how a secret can be accessed and by who, and was then loaded into Conjur. After a variable was loaded into Conjur, it could be assigned the secret data, as seen in step 6. As with the other solutions, Curl was first used to verify the API call, which was later implemented as a wrapper class for the factory in the test application. The wrapper class was then used for unit tests and benchmarking.

E.3 Implementation steps

1. Download Conjur tutorial repository and change directory.

```
$ git clone https://github.com/cyberark/conjur-tutorials.git
$ cd conjur-tutorials/nginx
```

2. Run the installation script that will create the necessary Docker containers and perform basic configuration. Save the admin API key that is echoed to a secure location, such as a physical paper in a physical vault.

```
$ ./start.sh
```

3. Create a policy object for the host and the keys to keep secure and load it into Conjur.

```
$ cat conjur.yml
- !policy
  id: testproject #Host

- !policy
  id: APIKey #Secret

...

$ conjur policy load --replace root conjur.yml
```

4. Fill the host policy object with data about the host and load it into Conjur. Save the API key for the host that is echoed, this will be used by the testproject to communicate to Conjur.

```
$ cat testproject.yml
- !layer

- !host testproject

- !grant
  role: !layer
  member: !host testproject

$ conjur policy load testproject testproject.yml
```

5. Fill each secrets policy object with type and access control policy and load it into Conjur.

```
$ cat <Secret ID>.yml
- &variables
  - !variable password

- !group secrets-users

- !permit
  resource: *variables
  privileges: [ read, execute ]
  roles: !group secrets-users

- !grant
  role: !group secrets-users
  member: !layer /testproject

$ conjur policy load <Secret ID> <Secret ID>.yml

...
```

6. Add a secret that should be kept secure in Conjur.

```
$  conjur variable values add <Secret ID>/password <Secret>
```