Bringing Your Own Key to Azure

Emil Viftrup Jepsen, 202204239
Jonas M. K. Iversen, 202204975

Bachelor Report (15 ECTS) in Computer Science
Advisor: Diego F. Aranha
Technical Advisors: Laszlo Moldovan, Paula Villa Martin (Unit4)
Department of Computer Science, Aarhus University
June 2025

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The widespread use of cloud infrastructure and growing regulatory requirements have made it increasingly difficult for organizations to control the encryption keys that secure their data. In collaboration with Unit4, we have developed a prototype of a Bring Your Own Key (BYOK) system that enables their customers to regain control over the encryption keys securing their data. Three proposed solutions have been analyzed based on security, operational effort, flexibility, and customer control. Our prototype is based on one of the proposals, addresses the shortcomings of the other proposals, and enables Unit4's customers to retain control over key management. The prototype's design incorporates cryptographic authenticity mechanisms, access control, and alert-based auditing to mitigate common threats. Our results indicate it is possible to build a flexible Bring Your Own Key (BYOK) system that enables customers to keep control of the key management process. However, we also identified limitations in achieving complete end-to-end authenticity due to Cloud Service Providers' lack of support for key origin verification.

Jonas Iversen and Emil Viftrup Jepsen,
Aarhus, June 2025.

Contents

Abstract **ii**

1 Introduction **1**

2 Case Analysis **3**
2.1 Case Description 3
2.1.1 Entities 4
2.2 Threat Analysis 8
2.3 Security Policy 10
2.3.1 Security Policy: Middleware 10
2.3.2 Security Policy: Azure Key Vault (HSM) 11
2.3.3 Security Policy: Rest of the System 11

3 Analysis of the Proposals **12**
3.1 Attack Surface 12
3.2 Security Isolation 13
3.3 Implementation Effort 13
3.4 Operational Effort (Unit4) 13
3.5 Operational Effort (Customer) 14
3.6 Flexibility 14
3.7 Customer Control 15
3.8 Discussion 15

4 Prototype Design **18**
4.1 General Architecture 18
4.2 Authentication and Authorization Design 19
4.3 Key Management Design 20
4.3.1 Generation of Key Encryption Key 21
4.3.2 Upload and Rotation of Customer Provided Key 21
4.3.3 Key Deletion and Recovery 24
4.4 Certificate Chain Design 25
4.5 Key Auditing Design 26
4.6 Limitations of Design 26

5 Implementation **28**
5.1 Development Process 28
5.1.1 Automated Unit Tests 28
5.1.2 Manual Tests 29
5.1.3 Code Review 30
5.2 Architecture of the Implementation 30
5.2.1 Simplifications in the Implementation 31

6 Conclusion **34**

Bibliography **37**

A The Technical Details **40**
A.1 Unit Test Case Example 40
A.2 Code Coverage Report 41
A.3 Attack-defense Tree 42
A.4 Manual Tests 43
A.4.1 Authentication 43
A.4.2 Upload 45
A.4.3 Rotate 49
A.4.4 Compromise 53
A.4.5 Alert 55

Chapter 1

Introduction

Developments in the current threat space have forced governments and regulatory bodies worldwide to introduce stringent cybersecurity regulations targeting critical infrastructure and essential service providers [12, 9]. Meeting the requirements that these regulations mandate presents new technical challenges. Companies that provide services to customers subject to these regulations must adapt their services to help them meet the latest requirements. This task becomes even more difficult when companies rely on cloud infrastructure for their services, as they are constrained by the solutions offered by Cloud Service Providers (CSPs). This report examines a case issued by Unit4, which has emerged due to the new compliance requirements their customers are subject to, including *C5:2020 CRY-03* [9] and *NIS2 Article 21(2), point (h)* [12, 13]. The case is therefore essential for Unit4, as it addresses an increasing demand for additional functionality in their services, as their customers require more control over their cryptographic keys to adhere to modern compliance requirements, e.g. [9]. Unit4 specializes in cloud-based Enterprise Resource Planning (ERP) solutions, providing its services to many organizations. They leverage the Azure cloud platform to deliver their services. Section 2.1 describes the case in more detail.

This project aims to develop a prototype system to determine whether it is possible to create a Bring Your Own Key (BYOK) system that provides Unit4's customers with control over their cryptographic keys. Unit4 has provided us with three proposed solutions, and the prototype is an implementation of the third proposal that aims to address and eliminate the disadvantages associated with the other proposals. Developed as a Proof of Concept, the system will explore whether customer-managed encryption keys can be integrated into Unit4's services flexibly and securely. The case is, in essence, a key management problem, and we will use the idea of BYOK to enable the customers to do this. BYOK is a cryptographic key management model. It allows cloud customers to maintain control over the cryptographic keys the CSP uses to encrypt their data.

Many CSPs offer data encryption, key management services, and mechanisms for BYOK. These services usually utilize encryption keys generated and controlled by the CSP, meaning cloud consumers must trust the CSPs, thereby incurring greater risk. The organizations in critical industries cannot incur this risk due to regulatory requirements [9, 12]. These organizations aim to continue reaping the benefits of cloud computing while ensuring they meet data protection and security requirements. These requirements have led to key management techniques that allow cloud customers to retain control over the encryption keys protecting their data, thus ensuring confidentiality, authenticity, and integrity. The paper by Kamaraju et al. discusses various approaches for cloud data protection [16], ranging from *Default CSP Protection*, where the CSP has complete control, to Bring Your Own Encryption *BYOE*, where the customer has complete control. Meanwhile, BYOK lies between these extremes and offers a compromise where the customer is given more control but isn't entirely in control.

BYOK has seen adoption in cloud computing and other areas, such as Industrial Internet of Things (IIoT) [23]. The case study by Ulz et al. offers insight into how BYOK was implemented in a different setting. It highlights that this is not only a problem concerning cloud computing, but also IIoT, which, like cloud computing, is a technology that is still evolving. The leading CSPs also provide mechanisms that allow their customers to provide their own keys; for example, Azure offers BYOK for Transparent Data Encryption (TDE) [7]. What is not available is functionality that enables the companies' customers to leverage the services CSPs offer to bring their own keys. This becomes necessary when Independent Software Vendors (ISVs) utilize a CSP to provide customer services. This means that the CSP will store the data of the cloud customer's clients. When these third-party customers, due to regulatory requirements, are required to secure their data, we arrive at a situation where the current key management systems offered by the CSPs are no longer sufficient. This project takes its starting point in this situation. It attempts to address how ISVs, utilizing services provided by CSPs, can enable their customers to secure their data without compromising their security.

We would also like to address the limited literature on this topic. Although there is ample literature on secure cryptography, the same cannot be said for key management. Most of the articles are technical specifications, which we think is due to an academic perception of key management as a purely practical problem. This is surprising, as 65% of IT Professionals see *Cloud Security* as the most pressing security discipline, and *Data Security* ranks third [22], which is closely related to this problem. In this project, we would also like to stress the importance of good key management as *"cryptography is a tool that turns all sorts of problems into key-management problems"* [1, p 203].

This report will analyze the case and the proposals set forth by Unit4 using a security-first approach. To asses the threats and security considerations relevant to the case we start by systematically reviewing the entities that make up the environment in Section 2.1. Thereafter, we will, in Section 2.2, use the knowledge gained to complete a threat analysis, and based on this, we will define a security policy for the system in Section 2.3. Using this and an operational understanding of the system, outlined by Unit4, we will analyze the three proposals in Chapter 3. We will hereafter focus on the third proposal and see whether it is possible to build a prototype that provides the needed level of security while living up to functional requirements. In Chapter 4, we will explain the prototype's design, where the security policy has dictated many design choices. We will follow up with Chapter 5, where we explain the implementation process. In Chapter 6, we will present some suggestions for future work and conclude our work.

We used Generative AI for augmented search, critical feedback, and to boost programming productivity during this project. Please see the attached declaration for more details.

Chapter 2

Case Analysis

This chapter introduces the case and defines the architecture in which our solution must be incorporated. This architecture is a simplified model of Unit4's actual ERP system. We describe the case as presented by Unit4 in Section 2.1. In Section 2.1.1, we analyze the different entities in the system to determine their roles and significance. Using our insight, we complete a threat analysis in Section 2.2, which we use to specify the system's security policy in Section 2.3.

2.1 Case Description

As mentioned in the introduction, the case we are working on is stated by the ERP provider Unit4, which uses Microsoft Azure to host its product. We therefore start this section by introducing some terminology relevant to the case. Unit4 utilizes Microsoft Entra, formerly known as Azure Active Directory (AAD), to manage access to its cloud components and as its Identity Provider (IdP). The main components of their architecture relevant for this project are their Azure SQL DB, Azure Key Vault, and Azure App Services. The SQL DB communicates with the Key Vault to enable Transparent Data Encryption (TDE), which protects against offline attacks by encrypting *data at rest* [19]. Unit4 uses Azure App Services to serve its software.

The case we are analyzing was initially proposed as follows:

> *Some of Unit4's customers have the regulatory requirement that they should be able to use their encryption keys for Transparent Data Encryption (TDE) of their tenant's Azure SQL database. This requirement brings the challenge that the customer is responsible for and in full control of encryption key lifecycle management (key creation, upload, rotation, deletion), key usage permissions, and auditing of operations on keys. Unit4 suggests that BYOK can be implemented in their system in the following three ways:*
>
> - ***Proposal 1:*** *Unit4 will invite customers through a Guest AD Account into the Unit4 Azure Subscription. The Key Vault/SQL Server instance will be isolated/dedicated to the customer, setting up Access Control/Role-Based Access Control (RBAC) to manage the encryption key.*
> - ***Proposal 2:*** *Leverage Azure's capability to support cross-tenant customer-managed keys for TDE.*
> - ***Proposal 3:*** *Unit4 and the customer will develop a mechanism to synchronize the key in the Unit4 Azure subscription with the customer vault without human intervention.*

The first solution proposal does not require introducing any new components, only the configuration of existing ones. The second solution proposal introduces new components, primarily on the customer side, as they must create their own Azure

Subscription with an associated Key Vault. The architecture of the solution is shown in Figure 2.1. This solution is not trivial, but comprehensive guides describe how to configure it [24]. We will briefly outline the solution to provide the reader an understanding of the proposal they can leverage in Chapter 3. Unit4 creates a *user-assigned managed identity* and a *multi-tenant application*, which links the two tenants. The client then installs this multi-tenant application in their tenant and grants it access to their Key Vault. This setup enables the SQL DB on the Unit4 tenant to utilize the encryption key stored in the customer's Key Vault for Transparent Data Encryption (TDE), using the multi-tenant application as the link [24]. We will return to discussing the advantages and disadvantages of each proposal in Chapter 3.

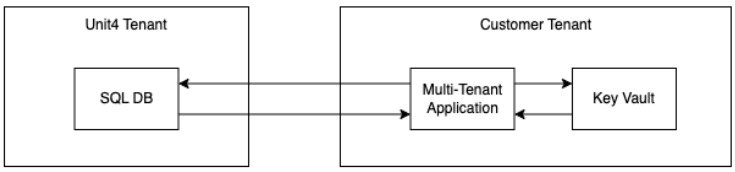

Figure 2.1: Network model of the second solution proposal

Following initial meetings with our advisor and technical advisors from Unit4, we successfully defined the architecture for the prototype system. This architecture is based on the third solution proposal and allows us to define the various entities that comprise the system (see Figure 2.2). The prototype system includes only the essential components necessary to understand the case and define the scope. We have chosen to base the architecture on the third proposal, as its implementation is not a question of configuration and must be implemented to determine whether it is a viable solution to Unit4's key management problem. Additionally, our preliminary analysis of the proposals showed that the third proposal offered the best trade-off between functionality and security. It is worth noting that the architecture for the first and second proposals is largely the same as that for the third proposal. With the exception that the Middleware is only present in the third proposal, and the second proposal also moves the Key Vault to the customer's tenant and introduces a multi-tenant application. We will now deliver a description of the different entities (E). We do this to give the reader an insight into the architecture and the case we are trying to solve. This analysis is also used to understand the system we are to develop. This understanding is the foundation for completing the subsequent threat modeling [18].

2.1.1 Entities

Entity: Azure Key Vault

The first entity is the *Azure Key Vault* (E1). This entity is crucial and has the **role** of storing the encryption keys and managing key operations. The *Azure Key Vault* has the following functions:

- Stores the encryption key that is used for TDE.
- Handle key management:

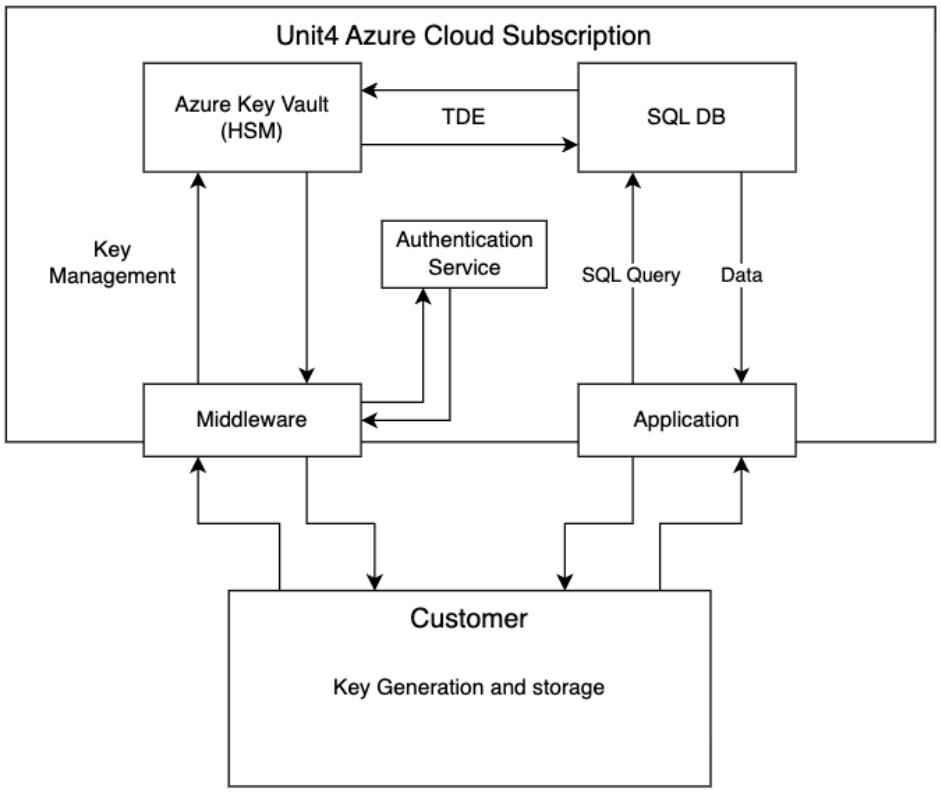

Figure 2.2: Network Model detailing the architecture of the prototype system for the third proposal

  - For customers using *Defualt CSP Protection* [16]: The customers without extra compliance regulations that let Unit4 handle key management.
  - For customers using BYOK: The Unit4 customers using this solution are primarily those with stricter compliance requirements, where the customers are required to handle key management. The customer may interact with the Key Vault to complete some operations, but they must fully control key management. In the BYOK context, the customer's keys are stored in the Azure Key Vault's underlying Hardware Security Module (HSM) and should not be accessible to Unit4 and Azure employees.
- Allows administrators to audit key usage.

We have also made the following assumptions about the *Azure Key Vault*:

- The Key Vault (Especially the HSM) is a Trusted Computing Base (TCB), which means that we assume it is secure and functions properly [1].
- The underlying HSM is a *FIPS 140-2 Level 2* or *FIPS 140-2 Level 3* validated HSM - depending on when the key is created. Azure's claims justify this assumption [4].

- Both the Key Vault and the HSM expose management interfaces.
- Assumed to be protected by a firewall, which separates the internal system components from the internet.

From the above description, it is clear that the Azure Key Vault communicates with the SQL DB. In the third proposal, the Key Vault also interacts with the Middleware when customers using BYOK need to complete key management operations (See Figure 2.2). This entity is crucial to the system's security.

Entity: SQL DB

The second entity is the *SQL DB* (E2), which has the **role** of storing (encrypted) data and responding to queries made by the Application. The SQL DB has the following functions:

- Configurable to use TDE.
- Stores the application's data.
- Responses to database queries issued by the Application.

We assume this machine is also protected from the internet by a firewall and has TDE configured to use an encryption key stored in the Key Vault. The SQL DB interacts with the Key Vault to encrypt data at rest and with the application to answer queries.

Entity: Application

The third entity is the *Application* (E3); this entity is the application that the customer is paying to use, and its **role** is therefore to serve that application. In the case of Unit4, the application is their ERP system. We use a dummy application, as its functionality is not the focus of this project. The application has the following functions:

- Provides customers with a UI and/or API to interact with the application.
- Submits queries to the SQL database.

We assume that this application utilizes the Authentication Service to authenticate users and can be accessed from the internet. The Application interacts with the SQL DB, the Authentication Service, and customers.

Entity: Customer

The fourth entity is the *Customer* (E4); we have chosen to consolidate the customer into a single entity that interacts with the system. The customer's **role** is to interact with and utilize the system. The customer has the following functions:

- Use the Application.
- Complete key management tasks (In the case where the customer uses the BYOK system).

When customers utilize their encryption keys for TDE through the BYOK system, they take control of key management, which means they must manage the associated risks and threats. We do not assume the customer is honest, as he may be compromised. The customer interacts with the Application and the BYOK system. In the first two proposals, the customer interacts directly with the Key Vault, while in the third proposal, they use the Middleware to complete key management tasks.

Entity: Authentication Service

The fifth entity is the *Authentication Service* (E5), whose **role** is to authenticate users. The Authentication Service has the following functions:

- Authenticate users.
- Access control.
- Logging of access.

We assume the Authentication Service is part of the system's Trusted Computing Base (TCB) and will behave properly. We also assume that this service is protected from the internet by a firewall and is thereby an internal component of the system.

Entity: Middleware

The sixth entity, which is only included in the third solution proposal, is the *Middleware* (E6). This entity corresponds to the *mechanism* that Unit4 addresses in their proposal. This entity has the **role** of acting as an intermediary between the Customer and the Key Vault. The Middleware has the following functions:

- Communicates with the Key Vault for key management on behalf of the customer. Supports all key management operations, including key upload, key deletion, key rotation, and key audit (The customer handles key creation on their infrastructure).
- Authentication of the customer using internal authentication mechanism (Using the Authentication Service).
- Logging of activity.
- Relay audit information to the customer.

This Middleware must be accessible by customers and, therefore, is not protected by a firewall. It is partly because of this that we do not assume the Middleware to be safe; we also make this assumption to see how much security we can provide with this Middleware, since this assumption forces us to follow the principle of least privilege. We also assume that the Middleware can set up communication using secure channels. Such that the traffic it receives from the customer is encrypted, and the traffic it sends to the Key Vault is also encrypted. The Middleware depends on the Authentication Service to ensure security.

2.2 Threat Analysis

With the fundamental understanding of the prototype system gained by defining the individual entities' roles, functions, and assumptions, we will now complete a threat analysis of the prototype system. To claim that a system is secure, we must specify the conditions under which it is secure, since it is impossible to make a system that is secure against all threats. It is essential to be aware of malicious actors' potential actions and tactics to achieve a secure system. It is also essential to understand the possible threats to the system. Identifying these threats can help us develop realistic and meaningful security requirements [18]. Also, implementing a system for the cloud setting means addressing new cybersecurity challenges that cloud computing introduces [16, 14], which makes it even more critical than it already is to complete rigorous threat analysis and set up a robust security policy.

It is essential to conduct a threat analysis as one of the first steps in the project. The security policy, which will be derived from this analysis, will lay the groundwork for system development. By considering these factors from the beginning, we can avoid the need to integrate security measures at a later stage. The functionality and features of the system should align with the security policy, rather than the other way around. Lastly, it is essential to proceed systematically. Otherwise, we risk leaving significant portions of the attack surface unaccounted for [18].

We completed a threat analysis [18] to highlight security features and vulnerabilities that must be addressed in our implementation. Our approach was inspired by Ulz et al [23]. This threat analysis begins with the system description, as presented in Section 2.1. We start here by listing the identified assets **(A)**, which are resources that must be protected from the adversary [18].

- (A1) *Encrypted Data*: Data at rest is encrypted and therefore confidentiality is ensured by some encryption key.
- (A2) *TDE Encryption keys*: The encryption keys that are processed at any of the entities during the BYOK process. Compromise or disclosure of the encryption keys would result in the loss of confidentiality and authenticity of the underlying data (A1).
- (A3) *Application functionality*: Introducing a BYOK system should not threaten the Application's functionality. It should not be possible for an attacker to launch an attack on the *Application* using the new BYOK system.
- (A4) *Key management system functionality*: The key management system for the customers not using BYOK should not be impacted, also the customers using BYOK should be able to perform their key management operations when they wish to.
- (A5) *Authentication service*: The introduction of the BYOK system should not affect the Authentication Service. It should not be possible for an attacker to launch an attack on the *Authentication Service* using the new BYOK system.

With the assets identified, we will now describe the attack surface in terms of its access points **(AP)**. When identifying the attack surface, we worked from an *adversarial* point

of view [1].

- (AP1) *Azure API*: The Azure management interfaces for the SQL DB, Key Vault, and other Azure Services.
- (AP2) *Application*: A malicious user can interact with the application to try and compromise all the above assets.
- (AP3) *Access to the hardware*: A malicious (insider) party with access to Azure facilities may have access to hardware ports.
- (AP4) *Configuration of Services*: This aspect of the attack surface involves the configuration of the different services. Misconfigurations can occur due to human error, creating opportunities for attackers to exploit these vulnerabilities. Additionally, attackers may be able to edit or inject malicious configuration files, which could be used to initiate an attack.

We utilized the characterization of the system and the identified assets and attack surface to determine the threats **(T)**, to which the system is vulnerable. For each threat, we have listed the affected assets:

- (T1) *Weak or buggy cryptography*: The encryption key is disclosed because of this.
  Assets affected: (A1), (A2)
- (T2) *Buggy implementations*: If there are bugs in the Application and/or Middleware, an attacker could exploit them to compromise the key management system and/or the Application.
  Assets affected: (A1), (A2), (A3), (A4)
- (T3) *Internet traffic encryption is misconfigured*: The attacker can eavesdrop on the channel and see the data transmitted in the clear.
  Assets affected: (A1), (A2)
- (T4) *Malicious insider actor*: When a Unit4 or Azure employee launches an attack from inside the system.
  Assets affected: (A1), (A2), (A3), (A4), (A5)
- (T5) *Faulty configuration*: The attacker exploits the fact that some configuration is not done properly.
  Assets affected: (A1), (A2), (A3), (A4), (A5)
- (T6) *DoS attack*: A malicious actor could complete a DoS attack on any entity accessible from the internet.
  Assets affected: (A3), (A4)
- (T7) *Supply chain attack*: An attacker could, for example, complete a dependency confusion attack.
  Assets affected: (A3), (A4)
- (T8) *Customer does not follow key management requirements*: The encryption keys are compromised at the source. Unit4 has to accept this threat, as the customer is

responsible for their keys when using BYOK [1].
Assets affected: (A1), (A2)

- (T9) *Social Engineering attacks*: The attacker tricks customer, Unit4, or Azure employees to compromise encryption keys, data, or credentials.
  Assets affected: (A1), (A2), (A3), (A4), (A5)
- (T10) *Replay Attacks*: The attacker eavesdrops on a channel and replays the messages he intercepts.
  Assets affected: (A1), (A2), (A3), (A4), (A5)

We have created an attack-defense tree [1], as shown in Figure A.3, to document the identified threats and attack surface. We used the attack-defense tree to analyze whether the system is susceptible to the outlined threats.

2.3 Security Policy

The threat analysis provided above helps us understand the threats that the system may face. This insight is beneficial in defining the system's security objectives. In this section, we outline the system's security policy, which is mainly geared toward the third proposal, as the Middleware is included. Section 2.1 serves as the specification of the system and is included as part of the security policy [1]. We have constructed our security policy by stating security objectives for the entity in question, where each objective is either something the entity is allowed to do, or something it is not permitted to do [1].

2.3.1 Security Policy: Middleware

It is vital for the system's security that only authorized users can interact with the Middleware. The Middleware should reject all users who cannot authenticate using multi-factor authentication (MFA) (SO1). Additionally, users should only be able to perform operations for which they have sufficient privileges (SO2). Thus, it should not be possible for an adversary or unauthorized party to perform key management operations.

It should also not be possible for a malicious party or the Middleware to impersonate a customer (SO3). Customers should also be able to audit their keys and the events related to them, and be alerted if changes occur (SO4). In Azure, the CSP and ISV can access the Key Vault; such access should result in an alert. The CSP or ISV should be unable to disable these alerts to obscure and hide their actions (SO5). Since this Middleware serves the customer, it should only be reactive (i.e., only performing the actions issued by a valid customer) (SO6).

There are also some security objectives regarding how the Middleware handles encryption keys. The cryptographic keys in transit to and from the Middleware should be sent via secure channels (SO7), such as two-way TLS. The Middleware should not

[1] If Unit4 wants to require the customer to adhere to key management requirements, then they need to create a contract that specifies this as a condition for using the BYOK system.

gain access to the secret keys in plain text form (SO8). Keys should never be stored in the Middleware (SO9), ensuring that if the Middleware is compromised, not even encrypted versions of the keys will be present.

2.3.2 Security Policy: Azure Key Vault (HSM)

The Key Vault is part of the TCB, and as such, it has special responsibilities. The Key Vault may only use the stored keys to wrap and unwrap keys for the SQL DB as part of the TDE process (SO11). All interactions with the customer's Key Vault during the BYOK process must be performed exclusively via the designated Middleware and only by authorized users. Key management operations — including key rotation, upload, or deletion must occur only after explicit initiation by the customer. No automated processes may manipulate customer keys without the customer's direct action (SO12). The Key Vault's underlying HSM should not use weak cryptography (SO13) and comply with FIPS 140-2 [20] (SO14). The Key Vault and the underlying HSM should not be able to leak keys to users with insufficient privileges (SO15).

2.3.3 Security Policy: Rest of the System

In addition to the security objectives described above, we have identified additional security objectives we want the rest of the system to uphold when introducing the BYOK process. Generally, we want the other system parts to remain functional and secure when introducing the BYOK system. These additional security objectives apply to the Application and the SQL DB entities.

The Application should still behave as intended and as it did initially. Introducing this new functionality should not compromise the Application's security and functionality (SO16). Likewise, the introduction of BYOK functionality should not compromise the security and functionality of the SQL DB (SO17). Furthermore, introducing the BYOK system should not result in the corruption or leakage of other customers' data (SO18). Lastly, there should be no increased risk for customers using the traditional key management approach.

Lastly, we *should* also consider the security objectives regarding the customer; however, due to the nature of BYOK, they maintain control over their keys. Customers should adhere to the key management policies they have established, but it is the customer's responsibility to define and maintain their security objectives, as the CSP and ISV have limited authority over them. Ultimately, the customer must ensure the security of their environment.

Chapter 3

Analysis of the Proposals

This chapter will examine and compare the three proposals based on eight parameters we have selected to assess the usability and security implications of each proposal. See Section 2.1 for a description of the three proposals. Table 3.1 summarizes the key trade-offs. We will now evaluate how each proposal performs with respect to each of the eight parameters: Attack Surface, Implementation Effort, Security Isolation, Operational Effort (Unit4), Operational Effort (Customer), Flexibility, and Customer Control.

3.1 Attack Surface

When discussing the attack surface, we focus on how much the attack surface will be expanded with the introduction of each proposal. The first proposal requires that the customer be given guest access to Unit4's AAD tenant, which means that an external user is given access to Unit4's AAD subscription. This is not inherently bad, but a strict access policy must be enforced to ensure this proposal is secure, such that customer users only have access to components required to complete their tasks. Even if precautionary measures are taken, the attack surface expands with each additional external user, since they can be compromised and used as an attack vector. Also, a larger amount of configuration adds the extra risk of Unit4 administrators introducing configuration errors (T5), which a malicious user can exploit. We therefore assess that the attack surface will increase substantially if the first proposal is implemented.

Regarding the second proposal, it is challenging to determine how much the attack surface would expand. There is a security risk that a malicious actor may try to compromise Unit4's Azure environment through the customer's tenant using the multi-tenant application. While this concretely means that Unit4's attack surface is expanded, Unit4 is not as attractive to an adversary anymore, as they no longer possess the Key Vault, which is a valuable asset to the adversary. Since the Key Vault is now located on the customer's tenant, they have the responsibility that Unit4 previously had. They must consider possible attacks and deploy migrations to prevent them. This is a trade-off where the customer is given more control over the keys, but the customer pays in increased risk and a wider attack surface. In conclusion, the Key Vault's attack surface is moved to the customer, and Unit4's attack surface is expanded to include the multi-tenant application. As a result, the attack surface is increased, though not to the same extent as the first proposal.

The third proposal introduces a whole new system component. Unit4's attack surface is thereby expanded to include the Middleware. The attack surface has significantly increased, but this increase is not as significant as the first proposal. The third proposal only introduces one new component, while the first proposal adds an arbitrary number of external users, who are not easily managed.

3.2 Security Isolation

Regarding security isolation, the first and third solutions are very alike, since the keys and data are stored on the same tenant controlled by Unit4, which does not fulfill the *Separation of Duties* principle [16]. We have assessed that the third proposal has better security isolation than the first proposal, since the customer controls key management from the Middleware. This proposal ensures that Key Vault auditing and alert mechanisms have been set up, so the customer is alerted to changes. Thereby serving as an effective mechanism to ensure that Unit4 and/or Azure cannot overstep their privileges, thus mitigating threat T4 to a greater extent than the first proposal. This means the customer does not have to place the same level of trust in Azure and Unit4, as they are given complete transparency into activities occurring on the key management infrastructure. Therefore, we assess that the third proposal has a medium level of security isolation, while the first proposal only has a low level of security isolation.

The second proposal also has a medium level of security isolation, since it does obey the *Separation of Duties* principle [16] to some degree, by ensuring that the security provider (Customer) is located on a different tenant than the data storage provider (Unit4). However, it is a problem that the keys and the data are still stored on Azure. This means that the proposal does not completely ensure the *Separation of Duties* principle.

3.3 Implementation Effort

The first proposal requires a low implementation effort, as it is solved by configuring the Azure environment. Configuration guides are readily available and can be used to set up external users and access policies [17]. The second solution proposal requires a medium implementation effort, as it requires the customer to set up an Azure environment, which can be time-consuming, especially in cases where the customer lacks Azure training. However, there are guides available on how to set up this solution [24] that ease the implementation process. The third proposal requires a high implementation effort since the Middleware must be developed from scratch. Time and money must be spent designing, coding, testing, and deploying a secure application. In terms of security, the third proposal introduces additional risks by potentially creating vulnerabilities during the development process (T2). This proposal is therefore not as straightforward as using features developed by a trusted vendor. Threat modeling, a robust security policy, and extensive testing are required to ensure a secure solution. The development approach should also be chosen carefully and match industry standards. Therefore, addressing potential vulnerabilities will require significant additional effort.

3.4 Operational Effort (Unit4)

All the proposals require substantial effort from Unit4 to operate the system. The operational effort required by Unit4 is high for the first proposal because they must complete regular audits of external users and ensure proper lifecycle management of external users. These steps are necessary to maintain a secure system. This additionally

requires designing and maintaining procedures for these tasks to ensure they are done consistently. The second proposal requires medium effort by Unit4. They must keep track of the customers' tenants and the associated multi-tenant applications that make the configuration possible. The third proposal requires an amount of operational effort that lies between the other two proposals, so we have rated it as requiring a medium-high level of effort. The reasoning is that, once in production, the API must be maintained, which requires development effort. We have assessed that this requires less effort than ensuring external users are handled correctly, but more than keeping track of multi-tenant applications, hence the medium-high level of effort.

3.5 Operational Effort (Customer)

When looking at the operational effort for the customer, the second proposal requires a high level of effort. This proposal requires the customer to set up and maintain an Azure environment. In cases where the customer is using Azure as their CSP, then one could argue that the effort required is far lower, as they only have to expand their current setup. We have placed the effort level as high because there is no guarantee that the customer uses Azure as their CSP, and this proposal may require them to make a substantial infrastructure addition. The operational effort is also negatively affected because key operations must be performed manually through the management interfaces.

The first proposal faces similar problems, as the customer may not have sufficient knowledge of Azure and will need extra training to perform key management operations. Another issue is that performing key operations becomes cumbersome and time-consuming. These downsides might not be significant for some customers, since key operations are often performed sparsely. Nonetheless, we assess the operational effort required by the customer for the first proposal to be medium.

The design of the Middleware, being an API, allows the customer to integrate it into their key management system as they see fit. Additionally, it does not impose any requirements on the customer's infrastructure, removing the need to train customers to use Azure. Based on this, we have assessed that the third proposal requires low operational effort from the customer.

3.6 Flexibility

The third proposal is geared toward Azure, but it could be extended to support other CSPs. This should be possible if they provide sufficiently robust APIs for their Key Vault counterpart, which can facilitate the required key management operations. The third proposal can also be extended to the individual customer's needs, since it is a custom-developed application. Lastly, this proposal can be integrated into any customer infrastructure. These are factors that make the third proposal highly flexible.

The first proposal does not require the customer to make any changes or additions to their infrastructure, but they are forced to use the management interfaces made available by Azure and allowed by Unit4. Customers cannot choose how they wish to operate

their key management. Therefore, we have determined that the first proposal offers medium flexibility.

Lastly, the second proposal offers the least amount of flexibility and generally gives the customer a low level of flexibility. This proposal requires the customer to set up an Azure environment, and thus, doesn't integrate with customers using a different CSP or an on-premise solution. It also has the same problem as the first proposal, as it requires the customer to use the management interfaces made available by Azure and allowed by Unit4. The second proposal thereby places restrictions on the infrastructure the customer must use and mandates how key management must be operated.

3.7 Customer Control

When discussing customer control, we focus on the customer's control over key management and their keys. Here, it is clear that the second proposal offers a high level of control to the customer, as the Key Vault is located on the customer's AAD tenant; thus, they have complete control over it, see Figure 2.1.

The third proposal offers the customer a medium level of control, since all key management is controlled from one place, and key management is delegated to the customer. The main problem is that the keys are still stored on the Azure infrastructure, which means Unit4 can access the Key Vault since it is in their subscription. The Unit4 administrators with adequate privileges can give themselves rights to view which keys are present, view information on the individual keys, complete key operations, and modify the operations a particular key is allowed to perform. A benefit of the Azure Key Vault is that administrators will not be able to see the private keys, as they do not leave the vault. This is a considerable risk that must be addressed, since it is clear that the proposal is vulnerable to insider attacks (T4) performed by highly privileged users from multiple organizations.

In the first proposal, customers can access their Key Vault and perform key management operations manually. It is also possible for them to upload the keys they have generated using their own Key Management System (KMS). Therefore, the first proposal faces the same problems as the third proposal. The main difference is that customers can only perform key operations using the Azure management interfaces permitted by Unit4. Although the same can be said of the third proposal, as the Middleware by construction only allows interaction with sanctioned Azure management interfaces. We have therefore assessed that the first proposal offers the customer the same level of control as the third proposal.

3.8 Discussion

As shown in Table 3.1, each proposal has advantages and disadvantages. The first proposal is likely the easiest to implement in the short term; however, the administrative burden it creates will make it difficult to maintain. Although it gives customers some control over their keys, the increased attack surface and lack of flexibility mean it is only appropriate when a few customers require BYOK functionality.

The second proposal demands more initial effort but provides the best security isolation among the three options. Additionally, it requires the least amount of operational effort from Unit4, as they only need to manage the multi-tenant applications. However, this proposal places the highest operational burden on the customer, a significant drawback. Another problem is that it isn't flexible and forces customers to use Azure, which they may be reluctant to do. This proposal, therefore, delivers some outstanding security guarantees, but it falters in flexibility and customer operational effort to such a degree that it may strain the relationship between Unit4 and the customer. This proposal is therefore most appropriate for customers who already use Azure as their CSP.

On the other hand, the third proposal requires substantial upfront effort. Still, it compensates with high flexibility and lower operational effort for Unit4 and the Customer compared to the first proposal. A high degree of flexibility is essential, enabling customers to manage their keys from a central location and complete operations in their preferred way. This is beneficial because key management can often be complex, and simplifying it can enhance security. This helps customers exercise responsible key management and will improve their security and productivity. Another benefit is the low level of operational effort required of the customer. This offers customers a better deal than the first and second proposals. The main drawback is that the keys are stored on Unit4's Key Vault, and this drawback must be addressed with security mechanisms. The third proposal is very suitable if many customers with diverse requirements require BYOK functionality, as this will justify the significant initial investment.

The third proposal offers unique benefits that the other two do not. In the following chapters, we will develop a prototype for the third proposal to assess whether a flexible and secure BYOK system can be created. This prototype addresses a significant drawback of the third proposal by incorporating additional security mechanisms that mitigate the threat of insider attacks (T4). It will also address the limitations of the two other proposals by being independent of the customer's infrastructure, not increasing the attack surface an intolerable amount, and not requiring an administrative burden that makes it hard to maintain.

Table 3.1: High-level comparison of the three BYOK proposals.

	Proposal 1: External Users in Unit4 AAD	**Proposal 2: HYOK on Customer Tenant**	**Proposal 3: Middleware BYOK**
Attack Surface	**High**: External users are given access to Unit4's AAD subscription	**Medium**: Customer takes responsibility for Key Vault; Unit4's attack surface is expanded by the multi-tenant application	**Medium**: Introduction of Middleware
Implementation Effort	**Low**: use built-in Azure flows; configuration only	**Medium**: set up of Azure environment; Creation of multi-tenant application	**High**: design, develop, test, and deploy secure Middleware layer
Security Isolation	**Low**: Keys and data are stored at the same tenant	**Medium**: Separation of which tenant the keys and data are stored on. Everything is still stored on Azure.	**Medium**: Keys and data are stored at the same tenant. Key management is addressed at another location
Operational Effort (Unit4)	**High**: lifecycle management & audit of external users; manual key management operations	**Medium**: setup/maintenance of multi-tenant applications	**Medium–High**: build, operate, secure, and version-manage custom Middleware
Operational Effort (Customer)	**Medium**: Azure training; Manual key management	**High**: Must maintain own Azure infrastructure; Manage multi-tenant integration; Azure training	**Low**: Integrate with API;
Flexibility	**Medium**: Customer can have any infrastructure; Must use Azure for Key Management	**Low**: Customer must create Azure infrastructure; Same procedure for each new customer	**High**: API can be adapted to the needs of the Customer and can also be extended to work with multiple CSPs
Customer Control	**Medium**: Unit4 has complete vault control; limited audit visibility	**High**: Customer has complete vault control; meets strict compliance requirements	**Medium**: Central key management; high audit visibility

Chapter 4

Prototype Design

When designing the system, we used the security objectives stated in Section 2.3 to ensure that the system has the desired level of security. The security objectives serve as the foundation for designing a secure system. If they are not satisfied, then the security guarantees are weakened. The design described in this chapter also considers the threat model outlined in Section 2.2 and details the security measures implemented to mitigate various threats.

In this chapter, we will specify the design of the prototype for the third proposal, which involves the introduction of the Middleware. In the section below, we will outline the general architecture. After that, we will outline the specific components of the design in their respective sections. Lastly, we will address limitations of the proposed system in Section 4.6.

4.1 General Architecture

The general architecture of the solution is as stated in Figure 2.2. We have focused on the interaction between the Azure Key Vault and Middleware, as the Middleware is the component we are introducing into the system. As stated in Section 2.1.1, the idea is that the customer interacts with the Middleware when they want to perform key management operations. The Middleware also ensures that customers add alerts that monitor the state of their Key Vault and allows them to audit the keys.

In addition to the SQL DB and the Application, the proposed solution consists of the following components:

A Premium Azure Key Vault - which allows for HSM-protected keys.

Depending on the choice of Key Vault, we are provided with several security guarantees and fulfill some of the stated security objectives. Azure states that their premium Key Vaults are FIPS 140-2 Level 2 [1] certified [4], which is a fulfillment of (SO14). Additionally, the Key Vault imposes restrictions on the types of keys it can create and store (for example, RSA keys must be 2048 bits long), thereby preventing the use of weak cryptography, which fulfills (SO13). The Key Vault is also just a vault, meaning it does not initiate automatic processes (unless instructed), thereby fulfilling (S012). To fulfill (SO11), it must be configured for TDE and nothing else. The Key Vault's access control ensures that only users with the correct privileges can access key information (SO15).

One could worry that errors may occur in the configuration, as a considerable amount of configuration is required to ensure the Key Vault is set up correctly to achieve the

[1] Their managed HSM is FIPS 140-2 Level 3 certified - and the solution could also be expanded to handle interaction with a managed HSM instead of a Key Vault

necessary level of security. To mitigate this risk, we propose using Infrastructure as Code (IaC) to specify how the Key Vault should be set up, ensuring the ISV can consistently configure it correctly.

One of the Key Vault's strengths is also its biggest weakness: it is just a vault. It does not provide a way to configure additional validation, for example, when importing a key. We will return to this problem in Chapter 6.

A custom-developed API functioning as the Middleware.

We have chosen that the Middleware should be an API without a user interface. This design choice enables administrators to incorporate the API into their tooling for key management. This means the solution can be used without human intervention when integrated into the customer's infrastructure. The API will be deployed on an Azure App Service, ensuring some of the security goals out of the box. Azure App Services ensure that all communication with remote management tools (Such as the SDKs and APIs we will be using) is encrypted [10] and thereby ensures (SO7) for communication between the Middleware and the Azure Key Vault. The App Service should also be configured to enforce HTTPS [2] to ensure that (SO7) is fulfilled between the Customer and the Middleware. The use of HTTPS also mitigates the risk of man-in-the-middle attacks.

An Authentication Service.

Here, it would be possible to use an Authentication Service that the ISV already has and uses for its other services. We have utilized two external Authentication Services to replicate this relationship and simplify the development process. We therefore use Google and Microsoft as our (external) Authentication Services. We have included multiple authentication providers, as it is common in deployed systems. Our security policy states that users should be authenticated using MFA (SO1); we have chosen to move this responsibility to the Authentication Service, such that it becomes their responsibility to enforce it [3]. We will describe the authentication process in more detail in the next section.

4.2 Authentication and Authorization Design

The authentication design consists of two elements. First, the user authenticates using one of the external Authentication Services, verifying who the user is. This results in the Authentication Service issuing an identification token that our Middleware then uses to check whether the given user should have access. In cases where the user is authorized to access the Middleware, they are issued a JSON Web Token (JWT) access token, which they then use to authenticate themselves in subsequent API calls. The flow is shown in Figure 4.1.

This approach means that there is no custom check of the customer's credentials, and therefore, there is no need to store, for example, their password. It also, as discussed

[2] Which can be done using a simple one-click configuration [10]

[3] This is realistic in an actual deployment - as one would use authentication services affiliated with the customer, where MFA can be enforced

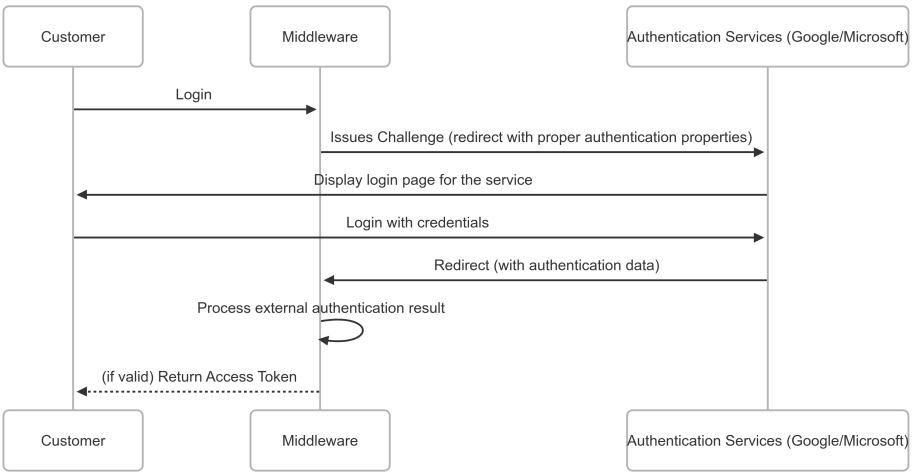

Figure 4.1: The authentication flow of the Middleware

in Section 4.1, has the advantage that MFA integration does not need to be set up from scratch. It is the responsibility of the Authentication Service to ensure that multi-factor authentication (MFA) is set up. Consequently, this also becomes the customer's responsibility, as they can control which Authentication Services are used and determine the rigor of the sign-in process. Thereby delegating (SO1) to the customer and the Authentication Service.

We must also ensure (SO2), which states that users should only be allowed to perform actions for which they have the necessary privileges. This can be achieved in multiple ways, for example, by using role-based authorization. We have adopted a simplified approach by creating a single group of users who have unrestricted access. Each user is identified by their unique email address. When issuing the JWT access token, the Middleware verifies whether the email address is permitted. Only users with allowed email addresses receive an access token. While this method does not offer the necessary granularity for a deployed solution, we implemented it in the prototype to ease development. Our recommendations for designing a robust authorization system in a production environment can be found in Section 5.2.1.

4.3 Key Management Design

To enable TDE for the SQL DB using a customer-supplied key, the following operations should be possible:

1. Generation of a Key Encryption Key (KEK) to be used for the upload process.
2. Upload of a customer-specified key.
3. Rotation of a customer-specified key.
4. Delete (Purge) a key (In case the key is no longer in use or compromised).

5. Recover a deleted key.

In designing and implementing these operations, we had to consider the behavior of the Key Vault and its specification. We utilized the management interfaces to interact with the Key Vault and perform the operations these interfaces allow. Additionally, we must adhere to the BYOK specification outlined for Azure Key Vaults [6], as it is the only way to upload keys.

We will now describe the design of the various operations and how the system's components interact to enable customers to perform them.

4.3.1 Generation of Key Encryption Key

The Key Encryption Key (KEK) encrypts the customer-supplied key, ensuring no unauthorized user can access its contents. It effectively ensures the confidentiality of the customer's key during transit to the Key Vault. The KEK must be an RSA key stored in the Key Vault's HSM [6], and in our design, it is a 2048-bit key (a larger key is also possible). The flow, as shown in Figure 4.2, illustrates the process of generating and issuing a KEK to the customer. When a customer requests a KEK, the request is forwarded to the Key Vault's underlying HSM (Azure-HSM), where it generates a KEK for key upload that is valid for 3 hours and returns it to the Middleware. The Middleware then encodes the KEK in the Privacy-Enhanced Mail (PEM) format and signs the tuple (KEK, PEM):

$$\sigma = S_{sk_A}(KEK, PEM)$$

Where sk_A is the Middleware's signing key stored in the Key Vault, and the Middleware requests that the Key Vault complete the signing on its behalf. The Key Vault then returns the signature σ to the Middleware, where the triple (KEK, PEM, σ) is returned to the Customer, who can validate that the key originated from the Middleware using the Middleware's public key pk_A. The customer must verify that the signature σ is a valid signature on the tuple (KEK, PEM). To enable the customer to perform this check, we introduce a certificate signed by a Certificate Authority (CA) that attests to the ownership of the signing key sk_A by the Middleware and make this certificate available to the customer. In this way, the customer can validate the certificate and use it to validate the signature. This ensures that the KEK received is actually from the Middleware, making it harder for an adversary to impersonate the Middleware. Section 4.4 provides a detailed description of this authenticity mechanism.

4.3.2 Upload and Rotation of Customer Provided Key

The processes for uploading and rotating a key are similar, as they both follow the BYOK specification [6]. The processes vary based on whether a Key Vault Alert check is performed, which verifies the presence of an alert based on the Key Vault's activity log. For each upload/rotation request, a new KEK must be generated first as specified in Section 4.3.1. We have designed the system to be flexible, allowing customers to choose how to upload their keys. They can upload the encrypted key material directly or use the transfer blob format [6]. See Figure 4.3 for a diagram detailing the key upload and rotation flows.

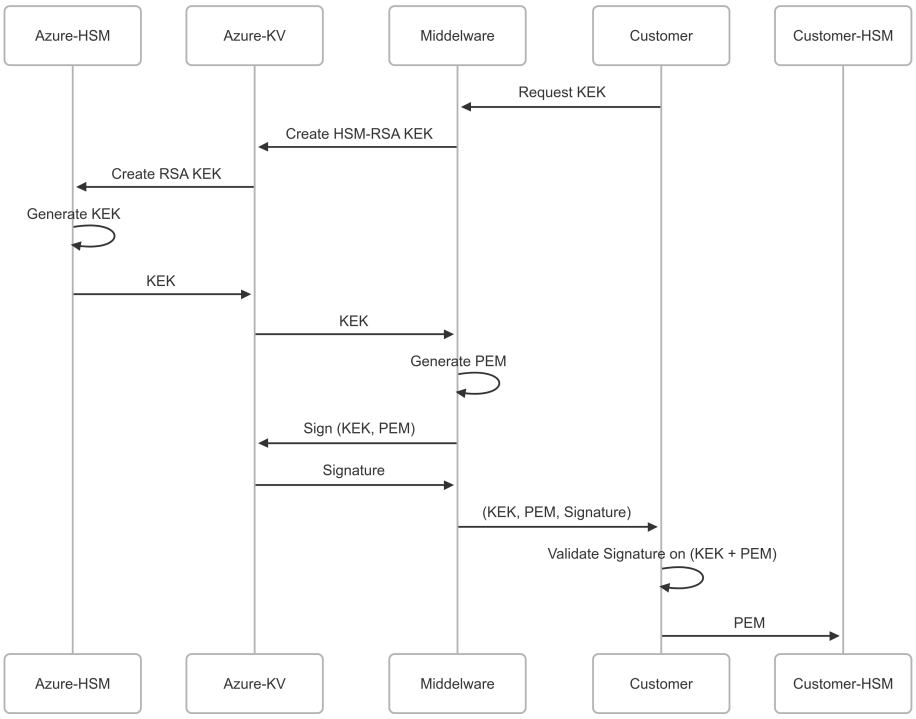

Figure 4.2: The interaction between the system's components to generate a KEK

When a customer wishes to upload a key, they first request a KEK for the upload process. They then use this KEK to encrypt their chosen key using the encryption mechanism "*CKM_RSA_AES_KEY_WRAP*" [8] and the algorithm *dir*, which means direct use of the shared symmetric AES key as the Content Encryption Key (CEK) [15]. Here, the content is the customer's chosen key. Therefore, the AES ephemeral key wraps the customer's chosen key. This ephemeral AES key is then wrapped using the RSA key received from the Middleware [8]. This is the algorithm and mechanism that the Key Vault uses for uploading keys [6]. When uploading their key, the customer must also provide a signature on the concatenation of a timestamp TS and the uploaded key material. Such that the customer uploads the following triple to the Middleware:

$$(KeyData, TS, S_{sk_C}(KeyData, TS))$$

Where *KeyData* can either be the encrypted key material or the transfer blob. Before forwarding the request to Azure Key Vault, the Middleware performs a few checks: it looks for a Key Vault Alert (this check is only conducted for upload requests), verifies the validity of the signature, and then constructs the request to be sent to the Key Vault using the *KeyData*. We will explain why a Key Vault Alert must be in place in Section 4.5. The Middleware validates the signature using the customer's public key pk_C and gains access to this key via a public key certificate that the customer provides. To ensure authenticity, this certificate must be signed by a Certificate Authority (CA) that the

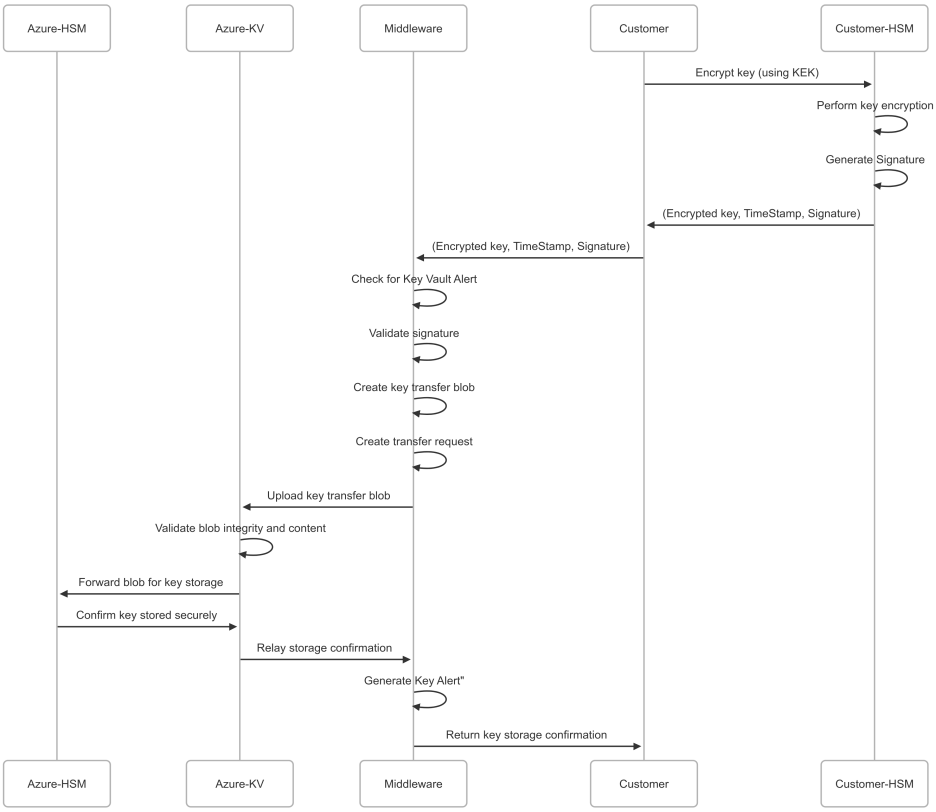

Figure 4.3: The flow for when a customer uploads a new key. The uploaded key material is the customer-provided key encrypted using the KEK. When the customer uploads a transfer blob instead, we make the following changes: The customer's HSM generates the transfer blob for the new key, and the Middleware therefore no longer needs to create the key transfer blob. Other than this, the process is the same. In both cases, if the Middleware fails in its validation steps, it stops execution and returns an error code to the customer.

In the case of key rotation, the process is very similar. The customer performs the same operations and uploads the key material to the Middleware. Here, the Middleware checks whether the given key exists and, if not, returns an error code, prompting the user to upload it instead. This ensures that a Key Alert is created the first time a key is uploaded.

Middleware trusts, and it must validate that the signing key belongs to the customer [4]. So, the Middleware not only validates the signature but also the customer's certificate. This mechanism has been added to the design to support authenticity and validate that the customer's key originates from the customer's key management solution and did not come from a malicious party impersonating the customer. It is thereby used to ensure (SO3).

The customer must also sign a timestamp to combat replay attacks (T10). This is added to prevent a malicious actor from replaying an intercepted key upload or rotation request to roll back to, for example, a compromised key. When the signature is validated, the Middleware also checks that the timestamp is not too old and, if so, refuses to upload the key material.

The Middleware forwards the request to the Key Vault after it has been validated. Upon receiving the key material from the Middleware, the Key Vault verifies whether everything is in order [5]. If there are no problems, it uploads the key to the underlying HSM and returns a confirmation that the key has been uploaded. Upon receiving this confirmation, the Middleware creates a Key Alert, which is an alert that triggers when the uploaded key is changed or viewed in the Key Vault. This Key Alert is only created the first time a key is uploaded and not in subsequent rotations. Lastly, the Middleware returns the upload confirmation to the customer.

Using the outlined design, the Middleware never gains access to the private part of either the KEK or the customer's key. Assuming that the Middleware behaves honestly, see Section 4.6. The key upload/rotation design thereby meets (SO8). Another feature of the Middleware's design is that the keys are never stored at the Middleware in any form, so that an attacker does not get access to key material by breaking into the Middleware, thereby ensuring that (SO9) is fulfilled.

Regardless of how the key is uploaded, we must ensure that a Key Vault Alert has been set up to notify the customer if changes are made to the Key Vault, including, among other things, privilege escalation and modifications to the Key Vault's configuration. Such an alert must be set up **before** a key can be uploaded, so the customer is notified of changes made to the Key Vault. When a new key is imported, the Middleware adds a Key Alert to monitor actions performed on that key, informing the customer of any changes made. Combining these two alert types ensures customers are notified if changes are made to their keys or the Key Vault containing them, as required by (SO4).

4.3.3 Key Deletion and Recovery

The design of the remaining key operations is relatively straightforward. These requests require forwarding the customer's request to the Key Vault, as if they were interacting directly with it. These operations must be included in the design and implementation, as they are necessary; the customer must be able to delete and/or suspend a key if there are signs of compromise. TDE requires that purge protection be turned on to ensure that data is not lost. As a result, it is not possible to purge keys that are stored in Key Vaults

[4] Preferably the customer's Hardware Security Module (HSM)

[5] This includes checking whether the key identifier refers to a key that can be used as a KEK

containing TDE protectors. Thus, when keys are deleted, they are actually soft deleted. In case of key compromise, the Customer should therefore rotate the TDE protector and soft delete the compromised key, where it will be purged after the retention period is exceeded.

4.4 Certificate Chain Design

With the authentication mechanism (Section 4.2), we ensure that the request comes from a valid user. We also want to ensure the customer can verify that a KEK originates from the Middleware. We can make this possible by having the Middleware sign the KEK, such that a malicious actor pretending to be the Middleware cannot complete the signature, thereby alerting the customer that something is wrong.

The customer must be able to verify this signature, which is achieved by creating a certificate that confirms the keyset (sk_A, pk_A) belongs to the Middleware and making it available to the customer. This certificate must be signed by a Certificate Authority (CA), allowing the customer to verify its validity and make it harder for an adversary to impersonate the Middleware. The customer can now verify the certificate with the CA. After validating the certificate, the customer can use the corresponding public key pk_A to verify the signature.

To provide the customer with access to the public key certificate, we will create an endpoint that enables them to download it, as shown in the interaction pattern in Figure 4.4.

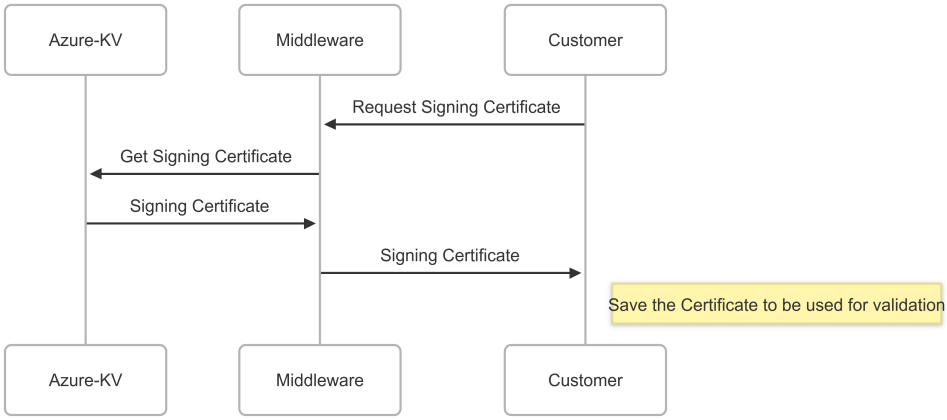

Figure 4.4: As can be seen, the certificate and its matching signing key are stored in the Key Vault. The Middleware uses the Key Vault's management interface functionality to sign items using the certificate's signing key. The signing key, therefore, never leaves the Key Vault and is also stored in the HSM.

We also require the customer to supply a certificate that confirms that the keyset (sk_C, pk_C) belongs to them, so that the Middleware can verify that it is a proper certificate and has been signed by a trusted Certificate Authority (CA). When the customer

uploads a key, the Middleware can use the certificate's public key to verify that the uploaded key material originated from the customer and not from someone impersonating them. This feature has been added to ensure (SO3).

4.5 Key Auditing Design

The design of key auditing should adhere to the security objective (SO4). That means the design should allow customers to audit the events in Azure that are relevant to their keys, such as changes to the Key Vault. We will explain how to request logs and utilize them alongside alerts for effective auditing.

Privileged users can request logs for various types of activities. Users can request information regarding the operations conducted on the stored keys and the actions taken within the Key Vault. They can also request activity logs. These logs include relevant administrative activities such as role assignments. To provide logs to the customer, the design includes Middleware endpoints specifically for this purpose. The customer can use these endpoints to request logs and gain insight into these activities.

Auditing can be performed at any time, but naturally, it will happen in response to an alert firing. These alerts are created as described in Section 4.3.2, and there are two different types: Key Vault Alerts and Key Alerts. Triggering on Key Vault activity and key activity, respectively. A Key Vault Alert must be set up before the customer can upload keys to ensure the customer is alerted if the Key Vault is tampered with. The Key Vault Alert is, thereby, a security measure to mitigate the risk of insider attacks (T4).

When an alert is triggered, customers can utilize the logging feature to review the events and take appropriate action. This ensures that the customer is notified if changes are made to the Key Vault or the stored keys. In the case where new role assignments are added to the Key Vault, the customer can validate and check the role assignments using an endpoint in the Middleware. This ensures that the customer has complete transparency into who has access to the Key Vault where their keys are stored, allowing them to take appropriate action if an insider attack occurs (T4). The design of the alerts and the auditing capabilities ensures that (SO4) is satisfied.

4.6 Limitations of Design

One of the significant problems with the design is that we need to trust the Middleware more than is appropriate for an externally facing application. The reason for this is that the Middleware has to complete many tasks that, in an ideal world, would have been handled by the Key Vault. These tasks must be moved to the Middleware because the Key Vault is merely a vault; it lacks additional functionality. There is no way for the Key Vault to validate that a customer uploaded key actually originated from the customer. This indicates that while we protect against third parties trying to impersonate the customer, there is currently no way to prevent the Middleware from doing so. Likewise, the Key Vault does not have the functionality to automatically sign the KEK before exporting it to the Middleware. There is therefore no way for the customer to validate

that the KEK is stored in the Key Vault. Consequently, we cannot fulfill (SO3) because the Middleware can impersonate a customer.

For the above attack to be possible, the Middleware must be compromised. In this case, the attacker will have unrestricted access to the Key Vault via the management interfaces anyway, meaning the attacker will be free to perform any operations they wish. To address this issue, we must ensure that the Middleware has the minimum privileges necessary for the Middleware to perform its tasks, aligning with the principle of least privilege. With such measures in place, the attacker will be able to upload a new TDE protector, but will not, for example, be able to access the SQL DB using the Middleware as it will not have access to it. Thus, the encrypted data (A1) is not necessarily compromised as a direct consequence of the Middleware being compromised (A4).

Another limitation of our design is that the Azure Key Vault is hosted on the ISV's infrastructure. This means that the ISV can force access to the alerts and the Key Vault. The design ensures that access to the Key Vault will result in an alert; however, protecting the alerts becomes more difficult because the owner can again force access. Creating additional alerts for the original alerts doesn't make sense, as the same problem would persist. This entails a certain level of trust between the customer and the ISV. Therefore, the design does not entirely fulfill (SO5).

Chapter 5

Implementation

This chapter will give an overview of the prototype's implementation. We have implemented the prototype as a web API in C# .NET. The code relies heavily on the management interfaces made available by Azure, both the SDKs and APIs. A large part of developing the prototype has been understanding these interfaces and the Azure infrastructure.

The application's code can be found on the GitHub repository Azure-BYOK. A deployed implementation can be found at Middleware Implementation. [1]

The goal of this section is not to go into detail about the specifics of the implementation. Instead, we will focus on the development process, give a high-level view of the implementation's architecture, and touch on some of the simplifications made in the implementation.

5.1 Development Process

We have used testing in our development process to ensure that our implementation behaves correctly, has the intended functionality, and satisfies the security policy's requirements. Automated tests were designed to boost productivity by allowing us to consistently verify that the code functioned as expected, thereby enhancing our confidence in the solution. The automated tests were additionally used to improve product quality and reliability, and made it possible to run regression testing effectively [21]. We have supplemented the tests with code reviews on pull requests to ensure that we both have insight into all aspects of the code and that the code remains maintainable.

5.1.1 Automated Unit Tests

The introduction of automated unit test cases made it possible to complete regression testing as the prototype was being developed. The tests are split into two namespaces `Test.Infrastructure` and `Test.Controllers` where the names indicate the projects they are testing (See Section 5.2).

In the `Infrastructure` namespace, we have the unit tests that test the services and their interaction with the Azure interfaces. These are primarily unit tests and assess how the *unit under test* interacts with the *dependent on unit*, in this case, how the infrastructure code interacts with the Azure interfaces.

When testing the controllers, we again created unit tests by mocking the implementation of the underlying services. Here, the focus was on the logic in the controllers, especially

[1] Connect to *https://webapp-middleware-byok.azurewebsites.net/authentication/login?provider=Google* to use Google's Authentication Service to attain an access token for the application, which can then be used for subsequent requests.

the error handling. We wanted to focus on the controller implementations and not the interaction between controllers and services, which is the rationale for mocking the service implementations [2] and not completing integration tests.

We have focused on making our tests as readable and maintainable as possible, and they have therefore been written using many of the principles laid forth by [11] for test-driven development (TDD). Even though we haven't employed TDD during the development process, many principles, such as *Evident Test* and *Representative Data*, are still applicable since they focus on making quality test cases [11]. An example of a test case where the principles have been applied can be seen in Figure A.1.

We used the code coverage metric during development to quickly identify parts of the implementation that were not covered by tests, allowing us to detect untested code efficiently. This way of working with the code coverage tool has yielded a high test coverage score, about 80% on the production code, i.e., the `API` and `Infrastructure` projects. See Appendix A.2 to see the *Code Coverage Report* produced by the *Rider* IDE.

5.1.2 Manual Tests

The manual tests have been completed in an environment with the architecture shown in Figure 2.2, but where the Application is a simple Blazor Web Application that draws weather data from the SQL DB. We have specified the architecture using Terraform to ensure the environment can be deployed quickly and reliably. The Terraform file `TestEnvironmentConfiguration.tf` also documents the architecture and how the different components in Azure interact. We have added twelve manual tests covering the primary key management operations and whether auditing and logging are correctly configured. These tests have been specified using the same schema and can be seen in the appendix Section A.4. They are split into five categories:

- **Authentication** - System tests verifying the authentication flow works as expected. See A.4.1.
- **Upload of customer-chosen key** - System tests verifying the key upload flow works as expected. See A.4.2.
- **Rotation of customer-chosen key** - System test verifying the key rotation flow works as expected. See A.4.3.
- **Compromise of keys** - Systems tests verifying whether customers can react to key compromise. See A.4.4.
- **Alert and logging** - System tests that verify whether the auditing mechanisms (alerts and logging) are set up correctly. See A.4.5.

They act as system tests, ensuring that the entire system functions correctly. These tests can also be viewed as specifications for the prototype's use, detailing how customers can use it to handle their key management tasks, for example, uploading a specified key.

[2] We used the Moq library

5.1.3 Code Review

We have ensured that all implementation changes were reviewed by creating a pull request on each code addition and modification. The other team member then examined the pull request. GitHub's pull request functionality has facilitated this workflow. We have chosen to spend time on code reviews to enhance software quality and minimize defects. We have completed the informal and tool-supported *Modern Code Review*, where we have strived to follow the recommendations *Beyond Defects*, *Communication*, and *Understanding* [3]. A significant benefit of code review in this project was the knowledge sharing aspect, such that we both gained insight into nearly all aspects of the implementation. Code reviews also made it possible to avoid accidentally introducing deprecated dependencies and reduced the overall number of bugs. The code reviews were thereby used to mitigate the threat (T2) and partly (T5), as the Terraform configuration file has also been subjected to review. We have also been aware that code reviews do not catch as many defects as expected [3], and our extensive testing suite therefore supported our code reviews in identifying and fixing defects during the development process.

5.2 Architecture of the Implementation

The code is grouped into four projects:

- **API** - The API controller code makes the Middleware's functionality available via API endpoints.
- **FakeHSM** - A HSM test double implementation that has been used to test the interaction with the customer's HSM.
- **Infrastructure** - The main bulk of the code base, where the services that interact with the Azure management interfaces are located.
- **Test** - The test code, containing all the unit tests as described in Section 5.1.1

The **API** and **Infrastructure** projects contain production code, while **FakeHSM** and **Test** contain test code. We have used the *dependency injection (DI) design pattern* in our implementation. The dependency injection configuration is centralized in the API's startup code, found in `API/Program.cs` . Here, all service interfaces are mapped to their concrete implementation. This ensures that the dependencies can be swapped in case of changing requirements.

We have separated the key management (business) logic from the controller logic by placing each in an isolated project. This ensures the separation of concerns principle, reducing coupling between the different classes. We have also throughout our implementation followed the principles *Program to an interface, not an implementation* and *Favor object composition over class inheritance* [11], to ensure our implementation is as flexible as possible. This can be seen in Figure 5.1, where the controllers only depend on the service interfaces (Abstractions) and not the concrete implementations, which makes it easy to test the controllers in isolation, as dependencies on interfaces can be mocked. It also reduces coupling, making it easier to change services if required.

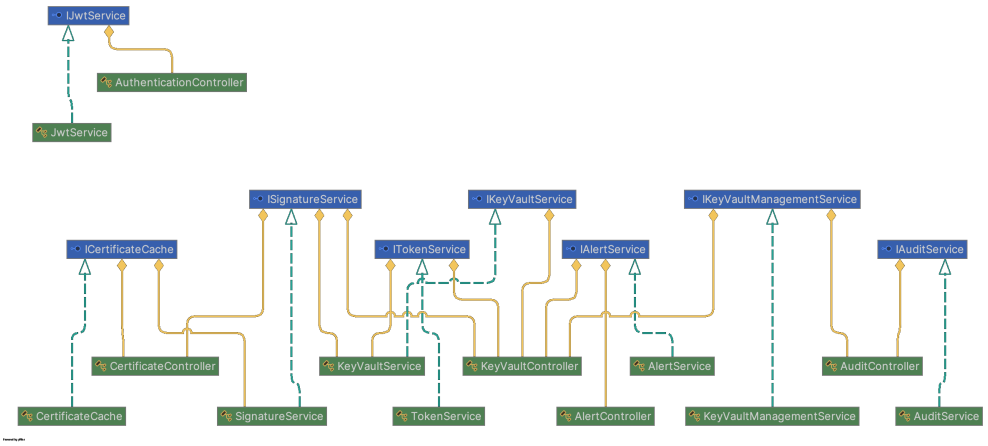

Figure 5.1: Type Dependency Graph for the Controllers and Services. Graph omits many helper classes, wrappers, and other functionality.

We have followed an error handling strategy in the controller code that catches exceptions, logs an error message, and returns appropriate HTTP responses. This ensures that errors are handled uniformly, and the log trace can be used to understand what happens when the code is in production. This observability has been expanded through structured logging throughout the application, which captures contextual information during the execution of each operation. These logs are available in the *App Service* when the prototype is deployed, allowing monitoring of the system and helping system administrators handle errors.

5.2.1 Simplifications in the Implementation

In some places, we have deviated from the design to simplify the implementation process. This was done because, with the project time frame, it wasn't possible to cover every aspect in the amount of detail needed for a production. We were therefore required to prioritize and focus our efforts to ensure all primary key management operations were supported and the prototype lived up to the security policy. We determined that the simplifications did not impact the prototype from addressing its primary concern, which was an attempt to see whether the construction of such a system was possible. The points discussed here can also be viewed as features that would need to be improved if the implementation were to go into production.

One of the places we have deviated is the certificate design, as we discussed in Section 4.4, it is essential that the certificates are signed by a Certificate Authority (CA). While this would be necessary for a deployed implementation, we have chosen to simplify the certificate implementation by using self-signed certificates, as this reduces the overhead of creating certificates. We assessed that using self-signed certificates was sufficient to highlight the overall idea and therefore chose to simplify the implementation in this way. In a production-ready implementation, the self-signed certificates would need to be replaced with certificates signed by CAs to give both the Customer and Middleware

the guarantee that the signing key belongs to the correct signer.

Another place where we have simplified the implementation is the process of uploading/rotating keys; a part of this process was the customer uploading a certificate to the Middleware, as outlined in Figure 4.3. This action is supported by the `CertificateCache` . The current implementation keeps only one certificate in the cache at a time. This means that if more than one user were to use the certificate functionality concurrently, they would overwrite each other's certificates. It is not inconceivable that multiple users will be working concurrently. Imagine a customer with multiple databases that need TDE protection, where the TDE protectors come from distinct HSMs. Therefore, if this implementation were to go into production, then this should be fixed, such that the certificate cache is updated to allow support for multiple certificates.

Another key simplification of this functionality is that certificates are merely cached and do not persist. Users must ensure the desired certificate is in the cache when uploading/rotating. This forces users to upload certificates an extra time just to be sure, which is problematic since only one certificate is cached at a time. Therefore, in a production-ready version of the implementation, it would be beneficial to allow persistent storage of customers' certificates, such that the user does not need to upload a certificate every time it needs to be used. This addition has the added benefit of making it straightforward to add functionality that enables the customer to manage the certificates that the Middleware uses, thereby giving the customer more control over the BYOK process.

The simplifications mentioned above were implemented because the single certificate approach simplifies the implementation without compromising its essential feature: providing authenticity. The way the certificate provides authenticity is unaffected. We therefore assessed that the single certificate approach was good enough for our prototype, and adding support for certificate management and persistent storage of certificates was not necessary to determine whether the conceived system was feasible and purposeful.

Another simplification in our implementation is the authorization mechanism. It uses an allowlist of allowed emails, which is insufficient for a production system. It could be improved by adding role-based authorization, such that different users could have different roles, for example, a *KeyManagementUser* role that can perform key management operations, a *LogAccessUser* role that can access the logs, and so on. This authorization system could be set up to be compositional, such that customers can use their preferred or mandated authorization schema. To allow such role-based authorization, the Middleware would have to be expanded to include a data layer, enabling it to keep track of the different users and their roles. We implemented the simple allowlist of emails in our solution since it fulfills the authorization requirement (SO2) in the simplest way possible. This approach removed the need to build a data layer, saving development time for more critical components. This highlights an area where the security policy is not strict enough, and it's essential to update it for a production system to enforce more stringent authorization requirements. Ideally, this should be done on a customer-to-customer basis to ensure that it specifies the authorization schema the customer wishes to use.

Lastly, the logs and alerts in our implementation are currently simpler than what would be ideal for a deployed implementation. One of the main challenges we faced during development was navigating the Azure Management Interfaces, which resulted in delays in implementing core functionalities. We encountered difficulties with how Azure manages logs in the Key Vault. In a production implementation, we would need to refine the log design to provide better clarity for customers. Additionally, customers should be able to query log data with finer granularity to avoid overwhelming them with information. Setting up alerts to activate correctly without triggering on regular activity also proved challenging, as they were closely tied to the log entries. We prioritized implementing key management functionalities over improving logging and alerts. However, a production implementation should enhance these areas to improve customer experience and security.

Chapter 6

Conclusion

This project aimed to develop a prototype that provides Unit4's customers with control over their cryptographic keys while eliminating the risk of introducing external users into the internal company environment, a risk associated with the first proposal. Additionally, we aimed for the prototype to provide the customer greater flexibility in integrating key management into their workflows and infrastructure compared to the second proposal. We implemented a Middleware system as an external extension of the Azure Key Vault to determine whether this was possible. We have demonstrated in this project that such a system is feasible; customers can perform the necessary key management operations and audit key usage via the Middleware. The Middleware, being an API, allows it to be flexibly integrated into customers' tooling and infrastructure. Most security objectives have been fulfilled to ensure the confidentiality and integrity of the customer's keys. Still, we have found that the prototype can not fully satisfy the ideal level of security proposed in the security policy of Section 2.3. We discovered that satisfying the security objective (SO3) was particularly challenging.

One of the obstacles to satisfying (SO3) was the Key Vault's lack of functionality:

1. It cannot verify the authenticity of a customer uploaded key.
2. It does not provide a mechanism for giving authenticity guarantees for its own generated keys.

During our project, Microsoft released key attestation for their Managed HSM service [5]. This would have made it possible for the customer to check that the Key Encryption Key (KEK) was created and is stored within the Hardware Security Module (HSM), thereby enabling the customer to verify the authenticity and integrity of the KEK. Choosing this functionality would necessitate a move from Azure Key Vault to Azure Managed HSM, which would require a rework of some of the implemented services. However, it could be interesting in future work to explore using a Managed HSM instead, such that this additional functionality can be utilized. We would also like to note that changing the underlying Azure Service would (only) require addressing the variability points between the Key Vault interface and Managed HSM interface, which could be done methodically using the 3-1-2 principle [11]. In general, we would expect that much of the code can be reused.

The key attestation functionality solves issue 2), but does not solve issue 1), and it is thereby still possible for the Middleware to impersonate the customer. To solve this, the Azure Key Vault would need to be expanded to provide a mechanism to check the authenticity of upload requests. A solution would require a mechanism for customers to provide a certificate for the entities allowed to upload keys, so that the Key Vault can check that the key originates from a valid source. Care would need to be taken to implement this, such that a malicious party can't just upload their certificate. This could be solved by asking a third party to attest that the uploaded certificate corresponds to

a valid entity's signing key, thus confirming that the certificate belongs to an entity allowed to upload keys. Solving this problem is not trivial, as there must be a validation mechanism to ensure the uploading entity is valid.

To further improve the security of the prototype, it would have been beneficial to use an API Fuzzer, such as restler-fuzzer [2], to find security and reliability bugs by fuzzing the API endpoints. We determined that this approach was unsuitable for our project because it would generate a high volume of API requests, potentially leading to operational costs that exceed our budget. Additionally, tracing the root cause of any errors identified by the fuzzer could be challenging. Given our limited knowledge beyond what is provided in the Azure documentation, it would be difficult to ascertain whether an error originated from our implementation or Azure itself. One could try to overcome these two issues by implementing a stub that can be used when fuzzing the API. However, even a rather involved stub might not sufficiently mimic the Azure services. We recognize that there are benefits to fuzzing the API, as it might discover security and reliability bugs vital to the usage and security of the Middleware. Something like this or some other dynamic software testing tool should be heavily considered before deploying an implementation to production. It would also be interesting to see which vulnerabilities an API fuzzer could uncover in future work.

Finally, it would be valuable and interesting in future work to see if a similar solution can be made with different cloud providers, such as an integration with AWS or Google Cloud. Whether the service corresponding to Azure Key Vault that these other cloud providers offer has the same functionality and allows for the same kind of implementation. From this, it could be interesting to find out which cloud provider's platform best allows for an extension of this kind. It could also be beneficial to determine whether a cloud-agnostic platform that supports multiple CSPs could be implemented. This would make it possible to supply BYOK functionality to many customers and make it easy for ISVs utilizing the cloud to change CSP while keeping the same BYOK system, thereby giving the customers and Unit4 the ultimate level of flexibility.

In summary, this prototype represents a significant advancement in enabling flexible and secure BYOK workflows. It also sheds light on ISVs' challenges in meeting their customers' compliance requirements. This project demonstrates that the current key management solutions offered by CSPs are inadequate for ISVs to address their customers' compliance needs effectively and highlights the shortcomings of these systems. We encourage Unit4 to carefully assess these gaps and work closely with their Cloud Service Provider, Azure, to strengthen their ability to deliver a system that meets their customers' regulatory requirements.

Acknowledgments

We would like to thank Unit4 for this great opportunity and collaboration. This naturally includes our technical advisors from Unit4: Laszlo and Paula. We want to thank Laszlo Moldovan for giving us insight into the architecture, completing code reviews, and guiding us within the Azure platform. Big thanks to Paula Villa Martin for helping us with the project process and report-specific considerations. Both provided us with invaluable feedback on this project and this report. We also want to thank Diego F. Aranha for advising us on this project.

Lastly, we thank the others who provided feedback and proofread this report.

Bibliography

[1] Diego F. Aranha, Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. *Secure Distributed Systems: An Introduction to Cryptography, IT-Security, Distributed Systems, and Blockchain Technology*. University of Aarhus, 2034.

[2] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE'19, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press.

[3] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[4] M Baldwin. About keys - azure key vault. `https://learn.microsoft.com/en-us/azure/key-vault/keys/about-keys#compliance`, May 2025. [Accessed 12-05-2025].

[5] M. Baldwin and chen karen. Validate azure managed hsm keys with key attestation. `https://learn.microsoft.com/en-us/azure/key-vault/managed-hsm/key-attestation?tabs=linux`. [Accessed 22-04-2025].

[6] M. Baldwin, mikefrobbins, amitbapat, and jlichwa. Bring your own key specification - azure key vault. `https://learn.microsoft.com/en-us/azure/key-vault/keys/byok-specification`, Apr 2025. [Accessed 24-02-2025].

[7] Zoran Barać and Daniel Scott-Raynsford. Configuring transparent data encryption to bring your own key. In *Azure SQL Hyperscale Revealed: High-performance Scalable Solutions for Critical Data Workloads*, pages 171–186. Springer, 2023.

[8] Dieter Bong and Tony Cox. PKCS #11 Specification Version 3.1. `https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.html`, February 2022. OASIS Committee Specification Draft 01. Latest stage: `https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html`.

[9] BSI. Cloud computing compliance criteria catalogue – c5:2020. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/ComplianceControlsCatalogue/2020/C5_2020.pdf?__blob=publicationFile&v=3`.

[10] Cephalin. What is security in azure app service? - azure app service. `https://learn.microsoft.com/en-us/azure/app-service/overview-security`. [Accessed 22-04-2025].

[11] Henrik B Christensen. *Flexible, reliable software: using patterns and agile development*. Chapman and Hall/CRC, 2010.

[12] European Commission. Directive (eu) 2022/2555 of the european parliament and of the council of 14 december 2022 on measures for a high common level of cybersecurity across the union, amending regulation (eu) no 910/2014 and

directive (eu) 2018/1972, and repealing directive (eu) 2016/1148 (nis 2 directive). *Official Journal of the European Union*, 50:80, 2022.

[13] European Commission, Directorate-General for Communications Networks, Content and Technology. Commission Implementing Regulation (EU) 2024/2690 of 17 October 2024 laying down rules for the application of Directive (EU) 2022/2555 as regards technical and methodological requirements of cybersecurity risk-management measures and further specification of the cases in which an incident is considered to be significant with regard to DNS service providers, TLD name registries, cloud computing service providers, data centre service providers, content delivery network providers, managed service providers, managed security service providers, providers of online market places, of online search engines and of social networking services platforms, and trust service providers. https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32024R2690, October 2024. OJ L, 2024/2690, 18.10.2024.

[14] Nils Gruschka and Meiko Jensen. Attack surfaces: A taxonomy for attacks on cloud services. In *2010 IEEE 3rd international conference on cloud computing*, pages 276–279. IEEE, 2010.

[15] Michael B. Jones. JSON Web Algorithms (JWA). RFC 7518, May 2015.

[16] Ashvin Kamaraju, Asad Ali, and Rohini Deepak. Best practices for cloud data protection and key management. In *Proceedings of the Future Technologies Conference*, pages 117–131. Springer, 2021.

[17] Csilla Mulligan. Add b2b collaboration users - microsoft entra external id. https://learn.microsoft.com/en-us/entra/external-id/add-users-administrator, Apr 2025. [Accessed 19-05-2025].

[18] Suvda Myagmar, Adam J Lee, and William Yurcik. Threat modeling as a basis for security requirements. *Symposium on requirements engineering for information security (SREIS)*, 2005.

[19] Pietervanhove. Transparent data encryption for sql database, sql managed instance, and azure synapse analytics. https://learn.microsoft.com/en-us/azure/azure-sql/database/transparent-data-encryption-tde-overview?view=azuresql&tabs=azure-portal, 2024. [Accessed 24-02-2025].

[20] FIPS Pub. Security requirements for cryptographic modules. *FIPS PUB*, 140, 1994.

[21] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012.

[22] Thales. 2024 cloud security study. https://cpl.thalesgroup.com/cloud-security-research, 2024. [Accessed: 2025-02-19].

[23] Thomas Ulz, Thomas Pieber, Christian Steger, Sarah Haas, Holger Bock, and Rainer Matischek. Bring your own key for the industrial internet of things. In *2017*

IEEE International Conference on Industrial Technology (ICIT), pages 1430–1435. IEEE, 2017.

[24] Pieter Vanhove. Cross-tenant customer-managed keys with transparent data encryption - Azure SQL Database & Azure Synapse Analytics — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/azure-sql/database/transparent-data-encryption-byok-cross-tenant?view=azuresql, 2023. [Accessed 24-02-2025].

Chapter A

The Technical Details

A.1 Unit Test Case Example

```
[Test]
public async Task ShouldValidActionGroupsReturnOk()
{
    // Given an alert controller and a valid action group
    var expectedResult = Mock.Of<ActionGroupResource>();
    _mockAlertService.Setup(mock =>
    ↪  mock.GetActionGroupAsync(It.IsAny<string>()))
        .ReturnsAsync(expectedResult);

    // When I ask to get the action group
    var actionGroupName = "test";
    var actionGroup = await
    ↪  _alertController.GetActionGroup(actionGroupName);

    // Then it should return an Ok result
    Assert.That(actionGroup, Is.Not.Null);
    Assert.That(actionGroup,
    ↪  Is.InstanceOf<OkObjectResult>());
    // and it should be logged
    MockLoggerTestHelper.VerifyLogEntry(
        _mockLogger,
        LogLevel.Information,
        $"Getting action group {actionGroupName}");
}
```

Figure A.1: Example test case to highlight the structure of our test cases and the used principles

A.2 Code Coverage Report

Figure A.2: Code Coverage Report produced by *Rider* IDE

A.3 Attack-defense Tree

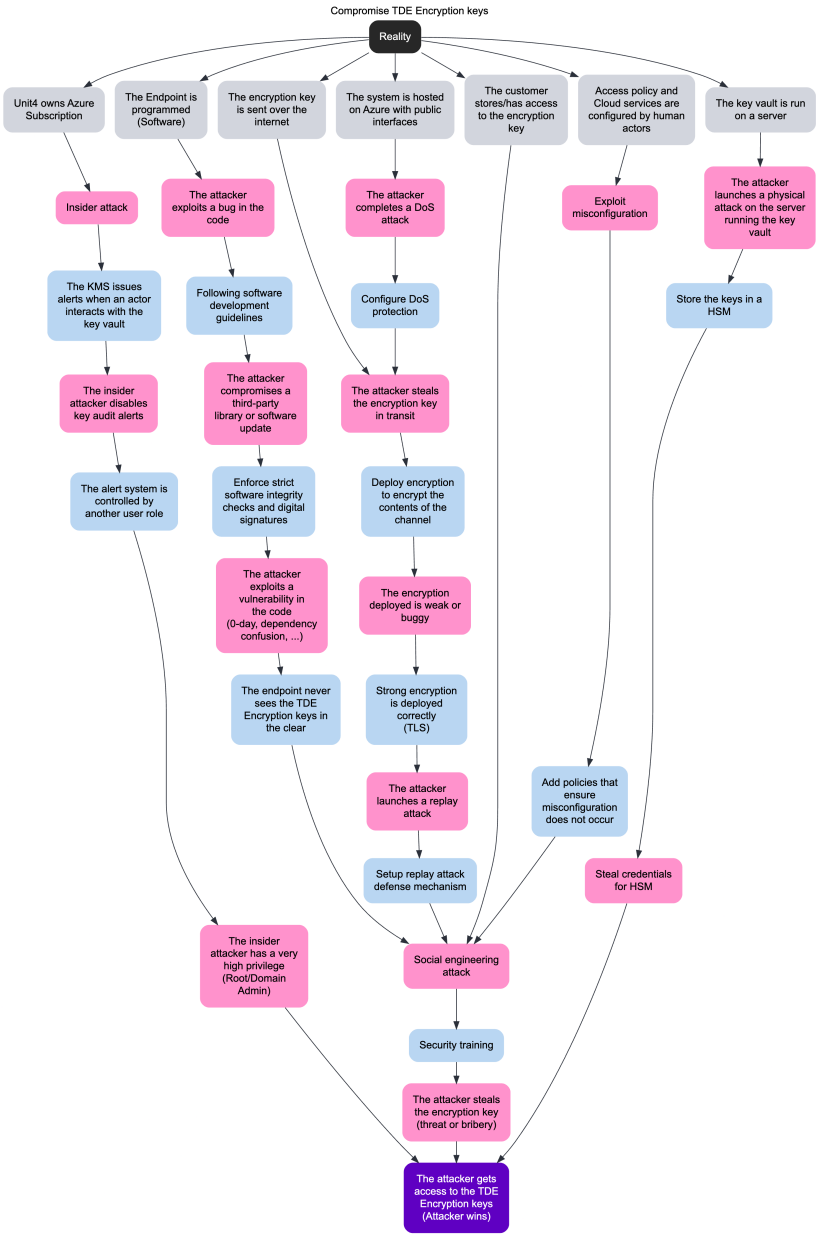

Figure A.3: Attack-defense tree documenting the threats against the BYOK system

A.4 Manual Tests

A.4.1 Authentication

Manual tests that verify the authentication flow works correctly and as expected.

Test Case ID: AUTH-001

Name: Authentication with Microsoft

Description: Verify that a valid user can authenticate via Microsoft Azure AD and receive a valid JWT access token.

Precondition:

- The environment is up and running.
- The test user's email is in the allowed list.
- The test user's email should not be a Personal Microsoft account.

Steps:

- Log in using the authentication provider's challenge. Go to: *https://webapp-middleware-byok.azurewebsites.net/Authentication/login?provider=Microsoft*.
- A page will open. Pick the correct Microsoft account to authenticate with and follow the prompted steps.

Expected Result:

- **Status Code:** 200 OK
- **Response Body:** includes a JWT `access_token`

Pass/Fail Criteria:

- **PASS:** The expected result matches the gotten result.
- **FAIL:** The authentication process fails.

Postconditions / Cleanup:

- See (AUTH-002) for the validity of the JWT

Test Case ID: AUTH-002

Name: Verify JWT `access_token`

Description: Verify the claims of the access token

Precondition:

- The environment is up and running.
- Successfully authenticated (see AUTH-001)
- Possesses a valid access token (Result of completing AUTH-001)

Steps:

- Copy the JWT access token to your clipboard.
- Navigate to *https://jwt.io*
- Paste the JWT access token into the input field.

Expected Result:

- **iss:** `https://webapp-middleware-byok.azurewebsites.net`
- **aud**: `https://webapp-middleware-byok.azurewebsites.net`
- **provider**: `"Google"` or `“Microsoft”`

Pass/Fail Criteria:

- **PASS:** The expected result matches the gotten result.
- **FAIL:** Any deviation.

Postconditions / Cleanup:

- N/A

Test Case ID: AUTH-003

Name: Use `access_token` to access the middleware

Description: Use the `access_token` to access the Middleware's other endpoints

Precondition:

- The environment is up and running.
- Successfully authenticated (see AUTH-001)
- Possesses a valid access token (Result of completing AUTH-001)

Steps:

- Copy the JWT access token to your clipboard.
- Navigate to *https://webapp-middleware-byok.azurewebsites.net/swagger/index.html*
- In the Swagger interface, click on the *Authorize* button.
- Enter the JWT access token.
- Click the *Authorize* button.
- Close the *Available authorizations* pop-up.
- Call the GET `/roleAssigments` endpoint.

Expected Result:

- **Status Code:** 200 OK

Pass/Fail Criteria:

- **PASS:** The expected result matches the gotten result.
- **FAIL:** Any deviation (Any status code other than 200 OK)

Postconditions / Cleanup:

- N/A

A.4.2 Upload

Manual tests that verify that the entire key upload functionality works as expected and correctly.

Test Case ID: UPLOAD-001

Name: Upload of Encrypted Key

Description: Test the complete process of uploading an encrypted key

Precondition:

- Successfully authenticated (see AUTH-001)
- Successfully authorized (See AUTH-003)
- The environment is up and running.

Test Data:

Parameter	Example Value
datetime	`08051608`
Action Group Name	`tg-0805`
Alert Name	`test-alert-0805`
KEK Name	`test-0805-1532`
Customer Email	`test.user@contoso.com`
time	`1608`

Steps:

1. **POST** `/group` with `"name": "tg-{time}"` and request body

```
{
    "members": [
        {
            "name" : "test-user",
            "email" : "test.user@contoso.com"]
        }
    ]
}
```

   - *Verify* 200 OK, response contains `name` and `"hasData" : true`.

- POST **`/alert`** - with `name = test-alert-{time}` and request body `["tg-{time}"]`
  - *Verify* 200 OK
- GET **`/Certificate`** - store the resulting certificate in a file named `cert`
  - *Verify* 200 OK

- GET **/KeyVault/create/{kekName}**: Generate a KEK with `kekName = test-{datetime}`, e.g. `test-09051329`. Save the entire response from **/KeyVault/create/{kekName}** to a file called `response-from-M.json`
  - *Verify* 200 OK
- Validate `base64EncodedSignature` using Python Script `local-cert-verify.py`.
  - *Verify* Script outputs *"The Signature is valid."*
- GET **/KeyVault/PEM/{keyName} :** With `keyName = test-{datetime}`
  - *Verify* 200 OK and response body is a key in PEM format
  - Save to file ("download")
- Upload the KEK PEM file into the local HSM.
- Use the HSM to generate a Key
- Encrypt the newly created key using the KEK and encryption mechanism `CKM_RSA_AES_KEY_WRAP`
- Use key management software to sign the concatenation of the newly created key and a timestamp using the customer's signing key.
- POST **/Certificate**: Upload customer's certificate to Middleware
  - *Verify* 200 OK. With response body: **Certificate was uploaded successfully**
- Post **/import/encryptedKey:** Where the request body contains the following attributes:

```
{
  "name": "test-c-key-{datatime}",
  "keyOperations": [
          "wrapKey",
          "unwrapKey"
  ],
  "timeStamp": "<timestamp>",
  "signatureBase64": "<signature>",
  "keyEncryptionKeyId": "<id>",
  "encryptedKeyBase64": "<encrypted_key>",
  "actionGroups": [
    "tg-{time}"
  ]
}
```

<timestamp> : timestamp that has been signed together with the customer's key. Timestamp in the following format: "2025-05-09T14:22:24.837Z"

<signature> : the signature that the HSM/KMS outputted

<id> : id of KEK used

<encrypted_key> : The customer's key in encrypted format protected by the KEK

  - *Verify* 200 OK

Expected Result:

- **Status Code:** 200 OK

Pass/Fail Criteria:

- **PASS:** All assertions above must be true.
- **FAIL:** Any deviation.

Postconditions / Cleanup:

- **DELETE**
  - `KeyVault/delete/{kekName}`
  - `KeyVault/delete/{test-c-key-{datatime}}`
- Delete the created action group in Azure
- Delete the created alerts in Azure

Test Case ID: UPLOAD-002

Description: Test the complete process of uploading a key transfer blob

Precondition:

- Successfully authenticated (see AUTH-001)
- Successfully authorized (See AUTH-003)
- The environment is up and running.

Test Data:

Parameter	Example Value
datetime	`08051608`
Action Group Name	`tg-0805`
Alert Name	`test-alert-0805`
KEK Name	`test-0805-1532`
Customer Email	`test.user@contoso.com`
time	`1608`

Steps:

1. **POST** `/group` with `"name": "tg-{time}"` and request body

```
{
    "members": [
        {
            "name" : "test-user",
            "email" : "test.user@contoso.com"]
        }
```

```
    ]
}
```

  - *Verify* 200 OK, response contains `name` and "`hasData" : true`.

- POST **`/alert -`** with `name = test-alert-time` and request body `["tg-{time}"]`
  - *Verify* 200 OK
- GET **`/Certificate -`** store the resulting certificate in a file
  - *Verify* 200 OK
- GET **`/KeyVault/create/{kekName}`**: Generate a KEK with `kekName = test-{datetime}`, e.g. `test-09-05-13-29`. Save the entire response from **`/KeyVault/create/{kekName}`** to a file called `response-from-M.json`
  - *Verify* 200 OK
- Validate `base64EncodedSignature` using Python Script `local-cert-verify.py`.
  - *Verify* Script outputs *"The Signature is valid."*
- GET **`/KeyVault/PEM/{keyName} :`** With `keyName = test-{datetime}`
  - *Verify* 200 OK and response body is a key in PEM format
- Upload the KEK PEM file into the HSM.
- Use the HSM to generate a key
- Encrypt the newly created key using the KEK and encryption mechanism `CKM_RSA_AES_KEY_WRAP`
- Use key management software to sign the concatenation of the newly created key and a timestamp using the customer's signing key.
- POST **`/Certificate`**: Upload customer's certificate to Middleware
  - *Verify* 200 OK. With response body: **`Certificate was uploaded successfully`**
- Post **`/import/blob:`** Where the request body contains the following attributes:

```
{
  "name": "test-c-key-{datatime}",
  "keyOperations": [
    "wrapKey",
          "unwrapKey"
  ],
  "timeStamp": "<timestamp>",
  "signatureBase64": "<signature>",
  "keyTransferBlob": {
    "schemaVersion": "1.0.0",
    "header": {
      "kid": "<id>",
```

```
      "alg": "dir",
      "enc": "CKM_RSA_AES_KEY_WRAP"
    },
    "ciphertext": "BASE64URL(<ciphertext contents>)",
    "generator": "BYOK v1.0; Azure Key Vault"
  },
  "actionGroups": [
    "tg-{time}"
  ]
}
```

<timestamp> : timestamp that has been signed together with the customer's key. Timestamp in the following format: "2025-05-09T14:22:24.837Z"

<signature> : the signature that the HSM/KMS outputted

<id> : id of KEK used

<ciphertext contents> : The customer's key in encrypted format protected by the KEK (Which must be base64 url encoded)

- *Verify* 200 OK

Expected Result:

- **Status Code** 200 OK

Pass/Fail Criteria:

- **PASS:** All assertions above must be true.
- **FAIL:** Any deviation.

Postconditions / Cleanup:

- **DELETE**
  - KeyVault/delete/{kekName}
  - KeyVault/delete/{test-c-key-{datatime}}
- Delete the created action group in Azure
- Delete the created alerts in Azure

A.4.3 Rotate

These manual tests verify that the entire key rotation functionality works correctly, and can be done directly after either of the upload tests in A.4.2, so the action group and alerts are in place. After completing the rotate test cases, complete the "Postconditions / Cleanup" from the upload test cases.

Test Case ID: Rotate-001

Description: Test the complete process of rotating a key using the new encrypted key

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Successfully uploaded a key (See UPLOAD-001 or UPLOAD-002)

Test Data:

Parameter	Example Value
datetime	`08051608`
KEK Name	`test-0805-1532`
Customer Email	`test.user@contoso.com`
time	`1608`

Steps:

- GET **`/KeyVault/create/{kekName}`**: Generate a KEK with `kekName = test-{datetime}`, e.g. `test-09-05-13-29`. Save the entire response from **`/KeyVault/create/{kekName}`** to a file called `response-from-M.json`
  - *Verify* 200 OK
- Validate `base64EncodedSignature` using Python Script `local-cert-verify.py`.
  - *Verify* Script outputs *"The Signature is valid."*
- GET **`/KeyVault/PEM/{keyName}`** **:** With `keyName = test-{datetime}`
  - *Verify* 200 OK and response body is a key in PEM format
  - Save to file ("download")
- Upload the KEK PEM file into the local HSM.
- Use the HSM to generate a key
- Encrypt the newly created key using the KEK and encryption mechanism `CKM_RSA_AES_KEY_WRAP`
- Use key management software to sign the concatenation of the newly created key and a timestamp using the customer's signing key.
- POST **`/Certificate`**: Upload customer's certificate to Middleware
  - *Verify* 200 OK. With response body: **`Certificate was uploaded successfully`**
- Post **`KeyVault/rotate/encryptedKey`** **:** Where the request body contains the following attributes:

```
{
  "name": "test-c-key-{datatime}",
  "keyOperations": [
    "wrapKey",
    "unwrapKey"
  ],
```

```
  "timeStamp": "<timestamp>",
  "signatureBase64": "<signature>",
  "keyEncryptionKeyId": "<id>",
  "encryptedKeyBase64": "<encrypted_key>"
}
```

<timestamp> : timestamp that has been signed together with the customer's key. Timestamp in the following format: "2025-05-09T14:22:24.837Z"

<signature> : the signature that the HSM/KMS outputted

<id> : id of KEK used

<encrypted_key> : The customer's key in encrypted format protected by the KEK

- *Verify* 200 OK

Expected Result:

- **Status Code:** 200 OK

Pass/Fail Criteria:

- **PASS:** All assertions above must be true.
- **FAIL:** Any deviation

Postconditions / Cleanup:

- **DELETE**
  - KeyVault/delete/{kekName}
  - KeyVault/delete/{test-c-key-{datatime}}

Test Case ID: Rotate-002

Description: Test the complete process of rotating a key using the new encrypted key stored in a transfer blob

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Successfully uploaded a key (See UPLOAD-001 or UPLOAD-002)

Test Data:

Parameter	Example Value
datetime	08051608
KEK Name	test-0805-1532
Customer Email	test.user@contoso.com
time	1608

Steps:

- GET **/KeyVault/create/{kekName}**: Generate a KEK with `kekName = test-{datetime}`, e.g. `test-09-05-13-29`. Save the entire response from **/KeyVault/create/{kekName}** to a file called `response-from-M.json`
  - *Verify* 200 OK
- Validate `base64EncodedSignature` using Python Script `local-cert-verify.py`.
  - *Verify* Script outputs *"The Signature is valid."*
- GET **/KeyVault/PEM/{keyName} :** With `keyName = test-{datetime}`
  - *Verify* 200 OK and response body is a key in PEM format
  - Save to file ("download")
- Upload the KEK PEM file into the HSM.
- Use the HSM to generate a key
- Encrypt the newly created key using the KEK and encryption mechanism `CKM_RSA_AES_KEY_WRAP`
- Use key management software to sign the concatenation of the newly created key and a timestamp using the customer's signing key.
- POST **/Certificate**: Upload Customer certificate to Middleware
  - *Verify* 200 OK. With response body :**Certificate was uploaded successfully**
- Post **KeyVault/rotate/encryptedKey :** Where the request body contains the following attributes:

```
{
  "name": "test-c-key-{datatime}",
  "keyOperations": [
    "wrapKey",
    "unwrapKey"
  ],
  "timeStamp": "<timestamp>",
  "signatureBase64": "<signature>",
  "keyTransferBlob": {
    "schemaVersion": "1.0.0",
    "header": {
      "kid": "<id>",
      "alg": "dir",
      "enc": "CKM_RSA_AES_KEY_WRAP"
    },
    "ciphertext": "BASE64URL(<encrypted_key>)",
    "generator": "BYOK v1.0; Azure Key Vault"
  }
}
```

<timestamp> : timestamp that has been signed together with the customer's key. Timestamp in the following format: "2025-05-09T14:22:24.837Z"

<signature> : the signature that the HSM/KMS outputted

<id> : id of KEK used

<encrypted_key> : The customer's key in encrypted format protected by the KEK

- *Verify* 200 OK

Expected Result:

- **Status Code:** 200 OK

Pass/Fail Criteria:

- **PASS:** All assertions above must be true.
- **FAIL:** Any deviation

Postconditions / Cleanup:

- **DELETE**
  - `KeyVault/delete/{kekName}`
  - `KeyVault/delete/{test-c-key-{datatime}}`

A.4.4 Compromise

These manual tests verify that a customer can react to compromised keys, and can be done directly after the upload test cases are completed, provided the action group and alerts are in place. After completing the compromise test cases, complete the "Postconditions / Cleanup" from the upload test cases.

Test Case ID: Compromise-001

Description: A key has been compromised; it must be deleted and purged. This test verifies that deleting and purging a customer-provided key is possible.

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Successfully uploaded a key (See UPLOAD-001 or UPLOAD-002)
- Purge protection is disabled on the Key Vault

Test Data:

Parameter	Example Value
key name	`test-c-key-{datatime}`

Steps:

- DELETE **`/KeyVault/delete/{keyName}`** where `keyName` is the name of the customer uploaded key
  - *Verify* 200 OK
- DELETE **`/KeyVault/purgeDeletedKey/{keyName}`** where `keyName` is the name of the customer uploaded key
  - *Verify* 200 OK

Expected Result:

- **Status Code** 200 OK

Pass/Fail Criteria:

- **PASS:** All assertions above are true (Note that if purge protection is enabled, that keys cannot be purged)
- **FAIL:** Any deviation

Postconditions / Cleanup:

- N/A

Test Case ID: Compromise-002

Description: A key has been deleted and must be recovered. This test verifies that it is possible to recover a deleted key.

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Successfully uploaded a key (See UPLOAD-001 or UPLOAD-002)

Test Data:

Parameter	Example Value
key name	`test-c-key-{datatime}`

Steps:

- DELETE **`/KeyVault/delete/{keyName}`** where `keyName` is the name of the customer uploaded key
  - *Verify* 200 OK
- GET **`/KeyVault/recoverDeletedKey/{keyName}`** where `keyName` is the name of the customer uploaded key
  - *Verify* 200 OK

Expected Result:

- **Status Code** 200 OK

Pass/Fail Criteria:

- **PASS:** All assertions above are true, and the customer uploaded key can be found in the Key Vault (Manual inspection)
- **FAIL:** Any deviation

Postconditions / Cleanup:

- N/A

A.4.5 Alert

These manual tests verify that the alert and logging functionality works as intended, and can be done directly after the upload test cases are completed, provided the action group and alerts are in place. After completing the alert test cases, complete the "Postconditions / Cleanup" from the upload test cases.

Test Case ID: Alert-001

Description: Test to verify that an alert is sent out when a key is tampered with and that the malicious activity can be viewed in the logs.

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Successfully uploaded a key (See UPLOAD-001 or UPLOAD-002)
- Access to the Azure Portal and the Key Vault
- Sufficient permissions on the Key Vault

Steps:

- Enter the Azure Key Vault using the Azure portal
- Find a customer uploaded key and disable it
- GET **`/keys/{numOfDays}`** where `numOfDays = 1`
  - *Verify* that there is a log entry detailing the disable action

Expected Result:

- **Status Code** 200 OK
- The members in the action group receive an email alerting them that a change has occurred.

Pass/Fail Criteria:

- **PASS:** The log entry is present, and the alert is received
- **FAIL:** Any deviation

Postconditions / Cleanup:

- Enable Key again

Test Case ID: Alert-002

Description: Test to verify that Key Vault tampering is visible in the logs.

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Access to the Azure Portal and the Key Vault
- Sufficient permission on the Key Vault

Steps:

- Enter Azure Key Vault using the Azure portal and add a new tag to the Key Vault. Something like: `name = test-tag , value = test-tag`
- GET **`/vault/{numOfDays}`** where `numOfDays = 1`
  - *Verify* that there is a log entry detailing the addition of the tag.

Expected Result:

- **Status Code** 200 OK

Pass/Fail Criteria:

- **PASS:** The log entry is present.
- **FAIL:** Any deviation

Postconditions / Cleanup:

- Revert the added tag

Test Case ID: Alert-003

Description: Test to verify that an alert is sent out when someone tries to escalate their Key Vault privileges, that the malicious activity can be viewed in the logs, and the changed role assignment can be viewed.

Precondition:

- The environment is up and running.
- Successfully authenticated (See AUTH-001)
- Successfully authorized (See AUTH-003)
- Access to the Azure Portal and the Key Vault
- Permissions to perform a role assignment

Steps:

- Enter Azure Key Vault using the Azure portal and create a new role assignment.
  - e.g., `Reader` role assignment
- GET **`/activity/{numOfDays}`** where `numOfDays = 1`
  - *Verify* that there is a log entry detailing the creation of a role assignment.

- GET `/roleAssignments`
  - The newly created role assignment should be visible.

Expected Result:

- **Status Code** 200 OK
- The members in the action group receive an email alerting them that a role assignment has been made.

Pass/Fail Criteria:

- **PASS:** The log entry is present, the alert is received, and the role assignment can be viewed
- **FAIL:** Any deviation

Postconditions / Cleanup:

- Revert role assignment
