Security in a Distributed Key Management Approach

Gunther Schiefer, Murat Citak,
Andreas Schoknecht
Institute of Applied Informatics and
Formal Description Methods (AIFB)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
e-mail: name.surname@kit.edu

Matthias Gabel, Jeremias Mechler
Institute of Theoretical Informatics (ITI)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
e-mail: name.surname@kit.edu

***Abstract*—Cloud computing offers many advantages as flexibility or resource efficiency and can significantly reduce costs. However, when sensitive data is outsourced to a cloud provider, classified records can leak. To protect data owners and application providers from a privacy breach data must be encrypted before it is uploaded. In this work, we present a distributed key management scheme that handles user-specific keys in a single-tenant scenario. The underlying database is encrypted and the secret key is split into parts and only reconstructed temporarily in memory. Our scheme distributes shares of the key to the different entities. We address bootstrapping, key recovery, the adversary model and the resulting security guarantees.**

Keywords-key management; key distribution; cloud security

I. Introduction

When personal data (e.g. medical records) is processed it is often imperative from a legal standpoint to guarantee confidentiality (e.g Data Protection Directive 95/46/EC, General Data Protection Regulation (EU) 2016/679 etc.). For many small and medium-sized enterprises cloud computing is an attractive alternative compared to operating an own data center. However, if sensitive data is handled by an untrusted third party like a cloud provider it is not trivial to ensure privacy of sensitive information.

The main adversary is an insider, e.g. a cloud administrator that has high privileges. To prevent a leak of personal identifiable data we use a database proxy This proxy acts as an adapter between the application and untrusted storage providers and encrypts all outgoing data transparently to the application side. We focus on using symmetric encryption (e.g. AES [3]) because speed is required for practicability. Independent of which outsourcing scheme is used, the encryption and decryption algorithm needs a secret key. This paper addresses the major issues of key handling and storage in a distributed system.

We assume that data in the cloud is persisted (stored) only in encrypted form but need to be processed unencrypted. Furthermore, we assume that all connections are secured on transport level with transport-layer security (TLS) [4]. The main focus of this paper is on handling the key used to encrypt the outsourced database as well as on authorization of the legitimate user.

To minimize the attack potential, the encryption key is broken into fragments and not persistently stored as a whole. The fragments are distributed between the application, the users and the proxy. The cloud storage provider is used only as a storage back-end and is not required to participate in the key management. Our key management scheme protects the database's confidentiality even if the key shares of an arbitrary number of users and the application are compromised and does not rely on heavy tools like a global PKI for user keys. This allows for a very lightweight deployment.

A. Related Work

One alternative to our approach is to store the secret key in the cloud itself. In this case the cloud provider is responsible for key storage and handling of encryption end decryption. The obvious upside is simplicity and transparency to the application. Amazon Web Services (AWS) [7] works like this; the user is not even able to obtain the secret key. In our model this approach is not applicable because we assume the application cloud provider cannot be trusted to handle plaintext data and the key. More along the lines of our work is the use of a key server which provides a trusted party like the database proxy with the encryption key when needed. However, this introduces the necessity of operating another component whose security must be ensured at all times [8] [9].

A completely different approach is called "Sealed Cloud", where encryption keys are stored at the cloud provider's premises, albeit in volatile memory only. By technical and organizational measures it is ensured that data and keys are erased as soon as (legitimate) administrative access to the server, possibly allowing to recover the key, is needed [10].

We build on top of an existing outsourcing scheme that relies on a single tenant key (TK). It was first introduced in the MimoSecco [1] and Cumulus4j [2] projects and further developed in PaaSword [5]. The key can be stored in secure hardware (MimoSecco [1]), on a dedicated key server (Cumulus4j [2]) or integrated (PaaSword [6]).

B. Contribution and Paper Structure

Our contribution is a simple, yet powerful distributed key management scheme that enables for transparent encryption of sensitive data. This scheme was introduced in a previous publication [6]. Here we detail the functionality of the components and define the needed interfaces. Additionally, we reconsider the establishment of trust during the bootstrapping

The original IEEE publication can be found here: http://ieeexplore.ieee.org/document/8104307/ ; DOI: 10.1109/CBMS.2017.151

phase and argue about the security of the scheme relative to an adversarial model we provide.

This work is structured as follows. Chapter II explains the underlying model for key management and introduces the entities that contribute to the scheme. The basic idea of key splitting and distribution is described. Chapter III explains in more detail how certain mechanisms, especially bootstrapping, key usage and recovery work. Chapter IV provides an attacker model and a security analysis. Chapter V concludes.

II. Key Management Model

The PaaSword approach is based on an architecture that separates the Application (A) where the data is processed from the DB Proxy (P) whose task is to store and access the data in a cloud database on behalf of A when authorized by a User (U_i) (see Figure 2).

The key management for data access shall avoid specific secure hardware. For security reasons, TK to access the database shall not be stored at P, where it would be beyond the tenant's control. On the other hand, the approach shall avoid the necessity of running a key server at the tenant side. Furthermore, TK shall not be available at A, so the application or its administrator cannot access the data at all times. In addition, no individual user should have TK, due to the high risk of losing the key or theft, especially if mobile devices (smartphone, tablet, laptop) are used. If a user had TK, he/she would be able to access the database directly; bypassing the access control mechanism of A if he/she got direct access to the cloud database. In addition, we need one key per tenant, to share access to the data to all users authorized by the tenant.

To fulfil all the requirements mentioned above, the approach of PaaSword is to split up the key in three parts and give one part to each U_i, one part to A and one part to P. Therefore, TK is split up in three parts TKu, TKa and TKp such that

$$\mathtt{TK} = \mathtt{TKu} \oplus \mathtt{TKa} \oplus \mathtt{TKp}$$

where $\oplus$ is the bit-wise XOR function. The user gets TKu, the application TKa and the DB Proxy TKp. Only if they work together they can reconstruct TK to access the database.

To have the ability to withdraw the access possibility for an individual user or to change the user part (TKu) of TK (e.g. it is lost or stolen) without affecting any other user it is necessary to have user individual triple sets of TK. Therefore, TK is split up for every individual U_i in a way that:

$$\begin{aligned}\mathtt{TK} &= \mathtt{TKu}_1 \oplus \mathtt{TKa}_1 \oplus \mathtt{TKp}_1\\ &= \mathtt{TKu}_2 \oplus \mathtt{TKa}_2 \oplus \mathtt{TKp}_2\\ &= \mathtt{TKu}_3 \oplus \mathtt{TKa}_3 \oplus \mathtt{TKp}_3\\ &= \ldots\end{aligned}$$

To create those user individual key triples, the tenant key TK is handled as a bit-string, the length of TK is l. For each user U_i, initially two uniformly random bit-strings, e.g., TKa_i and TKp_i , of length l are chosen. Then the third key, e.g., TKu_i is computed as

$$\mathtt{TKu}_i = \mathtt{TK} \oplus \mathtt{TKa}_i \oplus \mathtt{TKp}_i\ .$$

It is irrelevant which two keys are random and which is computed from the other two and TK.

A. Key Management Setup

To set up this scenario, the process shown in Figure 1 is used during bootstrapping time. P creates the encrypted database (with key TK), hands TK to the Tenant Admin and deletes TK from its memory. The Tenant Admin splits up TK for every user as described above. He keeps TK and all TKu_i in a safe place in case a recovery is necessary. Afterwards the Tenant Admin distributes the individual TKu_i to every U_i, all the TKa_i to A and all the TKp_i to P. The distribution to A and P is secured with transport encryption and two-sided authorization, the TKp_i are additionally encrypted and signed to prevent access or changes by A. When the setup is finished, the Tenant Admin should go offline to protect him from online attacks. He is only needed again in case of recovery or to add / change user.

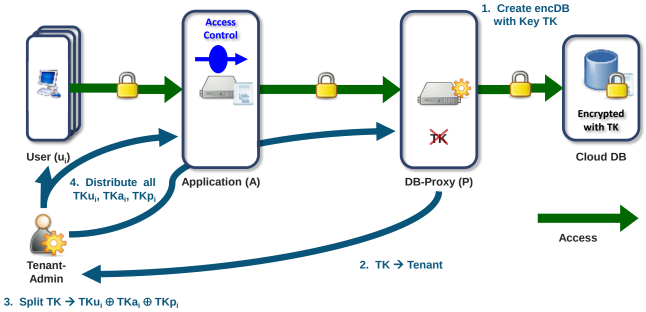
1. Create encDB with Key TK
Access Control
Encrypted with TK
~~TK~~
User (u_i)
Cloud DB
4. Distribute all TKu_i, TKa_i, TKp_i
Application (A)
DB-Proxy (P)
Access
Tenant-Admin
2. TK → Tenant
3. Split TK → $TKu_i \oplus TKa_i \oplus TKp_i$

Figure 1. Setup of the Key Management Mechanism [6]

B. Key Management During Runtime

Figure 2 shows the key management mechanism during runtime. U_i encrypts his individual part of the key (TKu_i) together with a timestamp using the public key of P (referred to as $Enc_P(TKu_i \| time)$) and adds it to his/her request to the application for processing data. A controls the permission of the user to process the data and if it is granted A can use $Enc_P(TKu_i \| time)$ and its own user specific part of the application key (TKa_i) to request the necessary database operations from P. P decrypts $Enc_P(TKu_i \| time)$ and controls the timestamp. If the timestamp is within the validity period, it reconstructs $TK = TKu_i \oplus TKa_i \oplus TKp_i$, checks the validity of TK and does the requested database operation. Afterwards P wipes TK out of its memory.

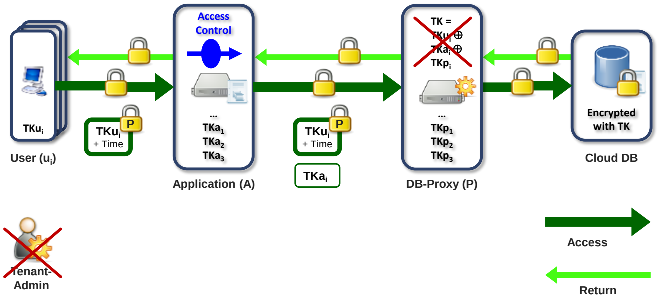

Figure 2. Key Management Mechanism during Runtime [6], modified

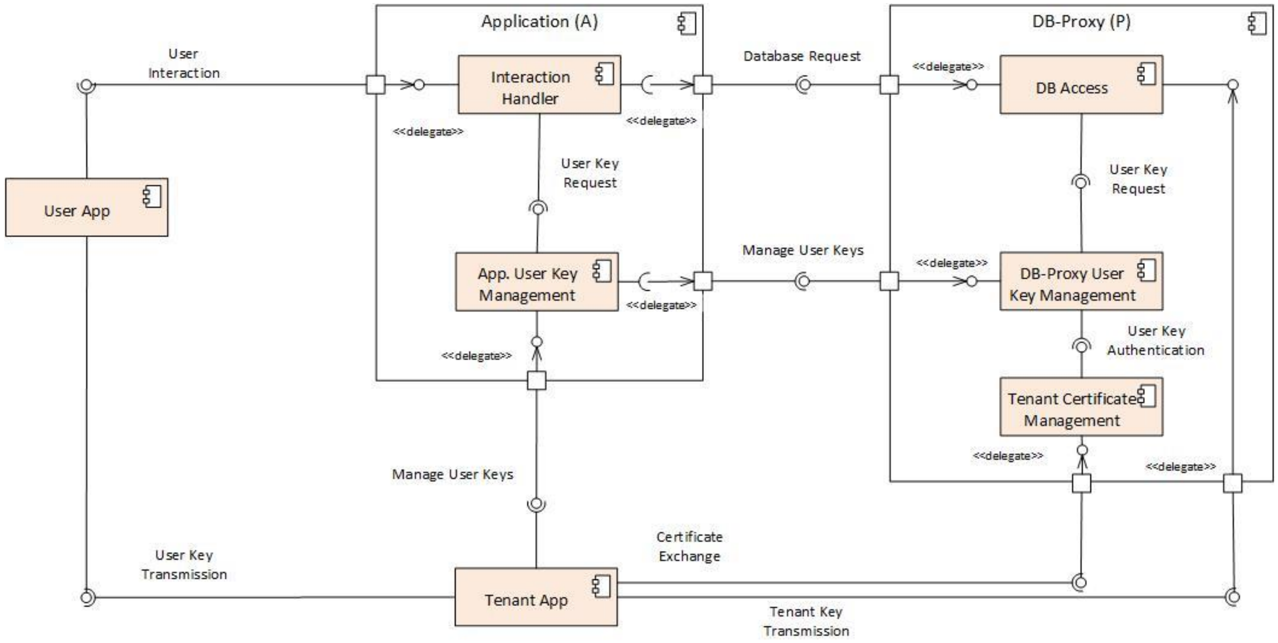

Figure 3. Key Management Component Diagram [6], modified

III. Components and Tasks of the Key Management Mechanism

The UML Component Diagram of Figure 3 depicts the main components that comprise the key management mechanism. These components are further elaborated below.

We assume that there is a (private or public) PKI for P and A. This also requires the possibility to perform a key rollover and revocation. Every instance (U_i, Tenant Admin, A, P) knows the public keys of P and A. Every connection between instances is encrypted and authenticated at the transport layer. For tenant isolation every tenant gets its own (virtual) DB Proxy (there is a 1:1 relation between tenants and DB Proxies)

A. Bootstrapping

We assume that the provider of A is also the service provider in contact with the customer (the tenant). Therefore, the provider of A initiates the process to bootstrap the Key Management. He creates a one-time password for authentication and orders a new P instance. The one-time password is added to the order.

The provider of the newly-created P creates a key pair for the new proxy instance and gets a certificate from the PKI and deploys it to P. Afterwards P creates a secret TK for the database and creates the encrypted database with TK. Using TK, P creates a ciphertext of the message "0" and stores it in the database for validation of the temporarily reconstructed TK during runtime. Furthermore, an additional signature key pair for the Tenant App is created by the Tenant Certificate Management.

Once P has been deployed, the provider of A notifies the Tenant Admin and passes him the one-time password (out of band!).

The Tenant App, authenticating itself with the one-time password, obtains the private signature key and the symmetric TK from P and stores them. Afterwards, TK is destroyed on P. The two interfaces between Tenant App and P are only needed once for bootstrapping and should be switched off afterwards.

B. User creation

We assume that every U_i has some code installed to add TKu_i to any request to A (called User App). This could be integrated in an application specific client or a local web proxy if a web browser is used as client application. The Tenant Admin uses the Tenant App to create a User-Id (UId_i), user individual keys TKu_i, TKa_i, TKp_i for every U_i and stores TK and all UId_i, TKu_i as described above. The user IDs UId_i and user keys TKu_i are distributed to every individual User App and all UId_i, TKa_i, TKp_i are handed to A (TKp_i timestamped, encrypted and signed). Therefore, the User Key Management within A offers an Interface for the Tenant App to add, change or withdraw a user and its keys. (Obviously the tenant admin has to authenticate with user credentials):

- AddUser (UId_i, TKa_i, $Sig_T(Enc_p(TKp_i \| time))$)
- ChangeKey (UId_i, TKa_i, $Sig_T(Enc_p(TKp_i \| time))$)
- WithdrawUser (UId_i)

The encrypted keys TKp_i for P have added a timestamp and are signed by the Tenant App to avoid manipulation by the A administrator. UId_i, TKa_i are stored at A. UId_i, $Sig_T(Enc_p(TKp_i \| time))$) are forwarded to the User Key Management of P where the signature is validated, $Enc_p(TKp_i \| time)$ is decrypted and the timestamp is validated before UId_i and TKp_i are stored.

The original IEEE publication can be found here: http://ieeexplore.ieee.org/document/8104307/ ; DOI: 10.1109/CBMS.2017.151

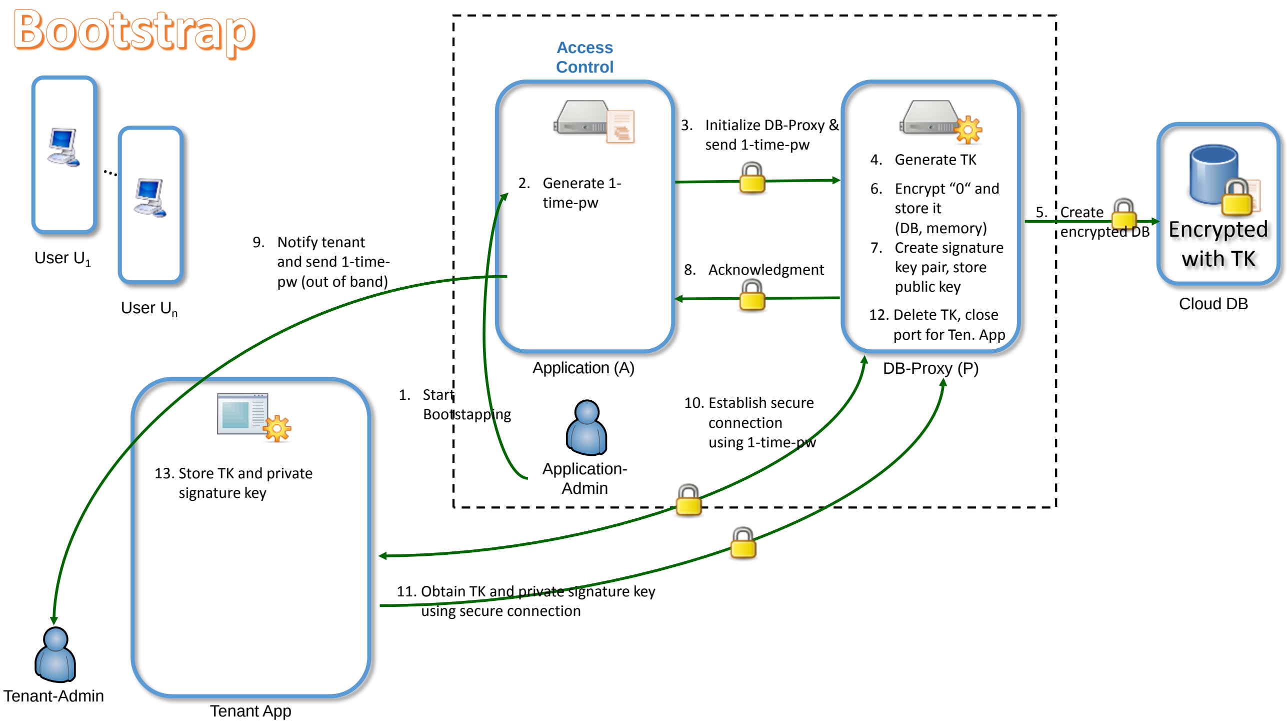
Bootstrap
User U1
...
User Un
9. Notify tenant and send 1-time-pw (out of band)
Access Control
2. Generate 1-time-pw
Application (A)
3. Initialize DB-Proxy & send 1-time-pw
8. Acknowledgment
4. Generate TK
6. Encrypt "0" and store it (DB, memory)
7. Create signature key pair, store public key
12. Delete TK, close port for Ten. App
DB-Proxy (P)
5. Create encrypted DB
Encrypted with TK
Cloud DB
1. Start Bootstapping
Application-Admin
10. Establish secure connection using 1-time-pw
13. Store TK and private signature key
Tenant App
Tenant-Admin
11. Obtain TK and private signature key using secure connection

Figure 4. Key Management Bootstrapping

C. *Key Usage During Runtime*

During runtime, every request to process data from a User App X to A has added UId_x, $Enc_P(TKu_x \| Time)$. To perform one User App request it can be necessary to perform more than one database request. This makes it necessary that the validation of a User App request is valid for a short timeframe. A adds UId_x, $Enc_P(TKu_x \| Time)$ and its part of the key (TKa_x) to each database request which is needed to perform the user request. P decrypts $Enc_P(TKu_x \| Time)$ and checks the timestamp. If the timestamp is within the validity period, it reconstructs $TK = TKu_x \oplus TKa_x \oplus TKp_x$, validates it against the stored (or cached) ciphertext of the message "0" and performs the requested database operation. Afterwards it wipes TK out of its memory.

D. *Key Recovery and Renewal*

In a distributed architecture several entities can lose keys or be corrupted. Our scheme can cope with such data losses and attacks as long as the tenant admin is not affected. The most likely case is that a user U_i loses his key part TKu_i. Since the tenant admin is physically near the user, recovery is very simple. After proper identification, the tenant admin gives the user a copy of TKu_i that he has stored.

If a user key TKu_i is compromised it is also very simple for the tenant admin to create a new set of key parts for user U_i replacing the old ones. Creation of new key triples is the same process as for the initial user creation, except that an existing UId_i is chosen and a different interface to A is used to indicate an update instead.

The only non-trivial case is when A or P gets compromised and recovered or loses user keys. The goal is to allow the users to keep their keys and only change the keys for the application and the proxy while TK remains the same. If it is necessary to have only new key parts for A and P the Tenant Admin is able to create new TKa_i and TKp_i with the Tenant App without the need to change the user keys TKu_i. They can be calculated by the knowledge of TK and TKu_i which are stored. The process is nearly the same as for the User creation, except that an existing UId_i is chosen, no new user keys TKu_i are created and distributed and the calculation process differs slightly.

IV. Adversary Model and Security Analysis

In the context of distributed database outsourcing with confidentiality and integrity (in the sense that an adversary should not be able to make changes to the outsourced database) as goals, security has to be considered at different places and layers.

A. *Protecting the Tenant Key*

The overall goal of our database outsourcing scheme is to protect the confidentiality of the outsourced data, which is achieved by only storing data encrypted with TK at the honest-but-curious cloud provider.

As P holds TK during operation which could, if an adversary had access to the encrypted database, completely break the database's confidentiality, several measures have to be taken in order to reduce the attack surface at P. As TK can be read from P's memory during operation, P should be deployed at a trustworthy location. If this is not possible, dif-

The original IEEE publication can be found here: http://ieeexplore.ieee.org/document/8104307/ ; DOI: 10.1109/CBMS.2017.151

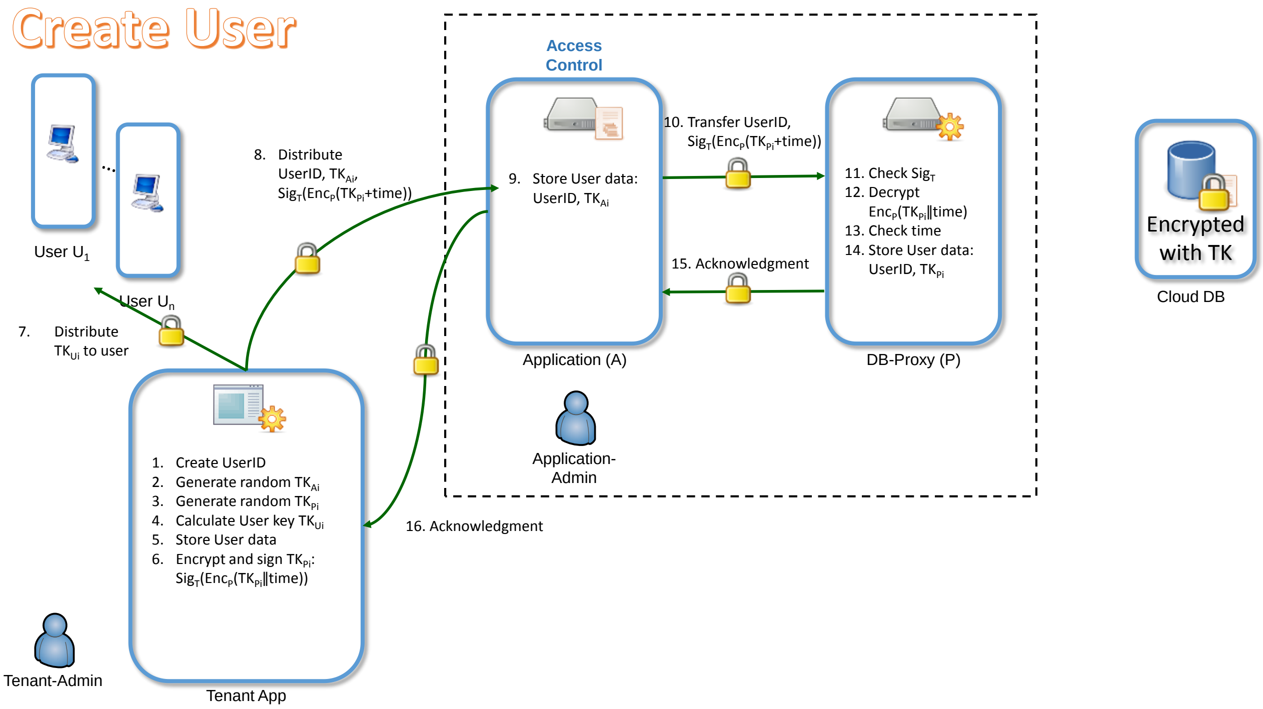
Create User
Access Control
User U1
User Un
7. Distribute TKUi to user
8. Distribute UserID, TKAi, SigT(EncP(TKPi+time))
9. Store User data: UserID, TKAi
10. Transfer UserID, SigT(EncP(TKPi+time))
11. Check SigT
12. Decrypt EncP(TKPi‖time)
13. Check time
14. Store User data: UserID, TKPi
15. Acknowledgment
16. Acknowledgment
Application (A)
DB-Proxy (P)
Application-Admin
Encrypted with TK
Cloud DB
1. Create UserID
2. Generate random TKAi
3. Generate random TKPi
4. Calculate User key TKUi
5. Store User data
6. Encrypt and sign TKPi: SigT(EncP(TKPi‖time))
Tenant-Admin
Tenant App

Figure 5. Key Management: User creation

ferent providers for the Cloud DB and P should be chosen as a honest-but-curious cloud provider hosting both entities could easily extract the key from P and use it to decrypt the outsourced database.

To minimize the possibility of key extractions, TK is only stored at P in volatile memory but never persisted. Furthermore, the key gets deleted after requests have been processed, protecting it during idle times. Consequently, the key is only vulnerable if P is compromised during query execution. Otherwise, TK is only persistently stored at the tenant.

In order to reduce the attack surface of the Tenant App and P, their only communication after the setup phase is through A. The Tenant App's commands are again protected with encryption, timestamps and signatures to prevent modifications and replay attacks. After the initial deployment, the Tenant App is not operational and therefore offline, except when changes to U_i, A or P keys have to be made. Even then, we only allow unidirectional communication originating from the Tenant App.

Due to the way TK is shared, even the simultaneous compromise of an arbitrary numbers of U_i and A does not leak TK as the user and application shares are information-theoretically independent from TK without the appropriate shares of P.

B. Bootstrapping Trust

Apart from considering attacks during operation, we also have to consider the bootstrapping process when the trust is first established. In particular, we have to consider adversaries at the network level, i.e. adversaries which try to replay or modify packets or imitate other protocol parties. While the Tenant App can verify P's identity by its certificate (assuming that the Tenant App knows the expected identity) for TLS, P has no way to verify the Tenant Admin's identity. We solve this by having A pass a one-time password out-of-band to the Tenant Admin, which can then be used for authentication during the bootstrapping phase.

During the bootstrapping phase, the Tenant App is provided with a signature key that can be used to authenticate its subsequent requests. Furthermore, the Tenant App is also given TK. This initial transfer is protected at the transport layer.

C. Protecting Information in Transit

During transit, sensitive non-key-related information such as user input, database queries and query results is transferred between the different entities. In order to protect such sensitive information during transit, we make use TLS throughout the protocol. When correctly used, TLS provides both confidentiality and integrity. Furthermore, it can be used to provide authentication of the communicating parties. In order to keep the deployment simple, we do not employ client certificates for TLS and thus only have authentication of the server (e.g. U_i or Tenant App can authenticate A at the transport level). Client authentication (e.g. authenticating U_i or the Tenant App from A's perspective) is handled by standard access control mechanisms such as passwords. In order to protect user-key-related information from A, communication is encrypted using the public key of P.

D. Protecting Against Malicious Protocol Parties

P can authenticate requests by checking if the accompanying key shares can be re-combined to a valid TK, implicitly authenticating the requesting U_i and A.

In the following, we assume that integrity and confidentiality are always provided at the transport layer and now consider compromised protocol parties, namely U_i and A.

In the context of A, confidentiality and integrity can be interpreted as enforcing appropriate user permissions within the application that ensure that U_i is only able to access records that he is supposed to and making only changes consistent with his permissions. Generally, this is done by standard access control mechanisms as well as user authentication, e.g. by username and password. If desired, accountability could be achieved by logging the user's actions. As a consequence, login credentials have to be protected both at the client and server level and during transit. By assumption, the latter is done at the transport level.

In order to provide stronger security guarantees (namely confidentiality of the database in the presence of a compromised U_i or A), relying on access control alone is insufficient as the party authorizing the user would have to be in knowledge of the whole secret. By distributing the secret between U_i, A and P, confidentiality of the database is ensured as long as at most any two instances are compromised, as the key cannot be recovered from two shares only. Unfortunately, splitting the user key does not protect the integrity if both U_i's login credential as well as his TKu_i are compromised as A cannot distinguish between an honest user and an adversary which is in possession of the correct credentials.

Due to the fact that the actual database query is created at A, U_i has no way to cryptographically protect his intended action. Thus, a compromised A is always able to take a TKu_i (even if it's encrypted) and use it to execute arbitrary SQL statements. This could in part be solved by moving part of the business logic to U_i, allowing him to verify the SQL statement created by A and sign it with a key known by P. However, this would break the overall application architecture and require the additional distribution of a signature key pair to the user resp. the certification of user-created keys, increasing the deployment effort, which contradicts the design goal of an easy setup procedure at the tenant's realm. Thus, this solution is out of scope.

As A only controls all TKa_i, an adversary compromising A has to either obtain a TKu_i or use the user's key-related input in an appropriate way to facilitate his actions. P's public key is distributed to U_i along with TKu_i, U_i is able to encrypt his share during transmission, protecting it from a possibly malicious A. In order to prevent replay attacks where A could try to use a ciphertext containing TKu_i multiple times, means such as counters or timestamps have to be taken. As we want to maintain as little state as possible at P, our approach makes use of timestamps whose validity can be easily checked. However, this requires synchronized clocks between U_i and P. Furthermore, there is an inherent delay between the creation of the ciphertext containing TKu_i and timestamp and the verification at P as the user's action has to be processed at A, possibly requiring numerous complex SQL statements. Thus, the timestamp has to be valid for a certain period of time after its creation, which opens an attack window for a malicious A.

V. Conclusion and Future Work

We presented a distributed key management scheme that does not rely on secure hardware and provides additional security compared to existing systems (e.g. AWS). There is no need to maintain a dedicated key server since all participating entities already exist in our generic scenario.

Future work will address extensions to our scheme that increase security. More fine-grained access by using table-specific keys will result in better privacy because an attacker can only see a fragment of the data if he obtains a secret key. The exploitation of secure hardware or trusted computing environments (e.g. Intel SGX) can further increase security.

Acknowledgment

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644814, the PaaSword project (www.paasword.eu) within the ICT Programme ICT- 07-2014: Advanced Cloud Infrastructures and Services.

References

[1] D. Achenbach, M. Gabel and M. Huber, "MimoSecco: A Middleware for Secure Cloud Storage," in *ISPE CE*, 2011.

[2] M. Huber, M. Gabel, M. Schulze and A. Bieber, "Cumulus4j: A Provably Secure Database Abstraction Layer," in *Security Engineering and Intelligence Informatics: CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, September 2-6, 2013. Proceedings*, Regensburg, Germany, 2013.

[3] V. Rijmen and J. Daemen, "Advanced encryption standard," in *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, 2001.

[4] T. Jager, J. Schwenk and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 V1.5 Encryption," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, Colorado, USA, 2015.

[5] R. Dowsley, M. Gabel, K. Yurchenko and V. Zipf, "A Database Adapter for Secure Outsourcing", in *2016 IEEE International Conference on Cloud Computing Technology and Science, CloudCom*, Luxembourg, 2016.

[6] R. Dowsley, M. Gabel, G. Hübsch, G. Schiefer and A. Schwichtenberg, "A Distributed Key Management Approach," in *2016 IEEE International Conference on Cloud Computing Technology and Science, CloudCom*, Luxembourg, 2016.

[7] "AWS Key Management Service (KMS)", Available online at: https://aws.amazon.com/en/kms/. Last Accessed 23-08-2016.

[8] Fraunhofer Institute for Secure Information Technology, "OmniCloud" Available online at: http://www.omnicloud.sit.fraunhofer.de/index_en.php, 2016. Last Accessed 19-06-2016.

[9] NightLabs Consulting GmbH. "Cumulus4j - Securing your data in the cloud - Deployment scenarios". Available at: www.cumulus4j.org/lateststable/documentation/deployment-scenarios.html. Last Accessed 23-08-2016.

[10] H. A. Jäger, A. Monitzer, R. O. Rieken and E. Ernst. "A Novel Set of Measures against Insider Attacks - Sealed Cloud". In D. Hühnlein(ed.), H. Roßnagel (ed.), Lecture Notes in Informatics - Open Identity Summit 2013, pp. 187–197, Gesellschaft für Informatik, Bonn, 2013.