Holistic Database Encryption

Walid Rjaibi
IBM Canada Lab, 8200 Warden Avenue, Markham, Ontario, Canada

Keywords: Databases, Encryption, Key Management, Security, Compliance.

Abstract: Encryption is a key technical control for safeguarding sensitive data against internal and external threats. It is also a requirement for complying with several industry standards and government regulations. While Transport Layer Security (TLS) is widely accepted as the standard solution for encrypting data in transit, no single solution has achieved similar status for encrypting data at rest. This is particularly true for database encryption where current approaches are forcing organizations to compromise either on the security side or on the database side. In this paper, we discuss the design and implementation of a holistic database encryption approach which allows organizations to meet their security and compliance requirements without having to sacrifice any critical database or security properties.

1 INTRODUCTION

Internal threats, external threats, government regulations, and industry standards require organizations to implement security controls to ensure information is adequately protected. Failure to do so can have a negative impact on an organization such as loss of customer data, damage to brand reputation, and even financial penalties. Encryption is a key technical control for protecting information. It is also an explicitly stated requirement for compliance with many regulations and standards such as the General Data Protection Regulation (Voigt et al., 2017) and the Payment Card Industry Data Security Standard (Chuvakin and Williams, 2009).

While TLS is widely accepted as the standard solution for encrypting data in transit, no single solution has achieved similar status for encrypting data at rest. This is particularly true for database encryption where current approaches are forcing organizations to compromise either on the security side or on the database side. Indeed, database encryption poses some very unique challenges as not only the solution needs to be sound from a security perspective, but it also needs to coexist in harmony with critical database properties such as performance, integrity, availability, and compression.

The rest of this paper is organized as follows. Section 2 discusses the related work around database encryption. In section 3, we state our contributions. Section 4 defines the threats our database encryption solution defends against. In section 5, we describe our solution design in full details. Lastly, section 6 summarizes our approach and outlines our future work.

2 RELATED WORK

Current database encryption solutions can be divided into four main categories: Column encryption (Benfield and Swagerman, 2001), tablespace encryption (Freeman, 2008), file system encryption (Anto, 2018), and self-encrypting disks (Dufrasne et al., 2016). Unfortunately, each of these solutions forces the organization to make a compromise either on the database side or on the security side.

Column encryption negatively affects database performance as queries with range predicates cannot benefit from index-based access plans to limit the data to read from the table. Instead, the database system is forced to read the entire table to evaluate the query. Tablespace encryption may leave certain data vulnerable to attacks when, for example, an administrator inadvertently takes an action that moves data from an encrypted tablespace to an unencrypted one. An example of such action would be the creation of a materialized query table (MQT) to speed up the execution of data warehousing queries. File system encryption and self-encrypted

Rjaibi, W.
Holistic Database Encryption.
DOI: 10.5220/0006833604720477
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 472-477
ISBN: 978-989-758-319-3

disks provide no protection against privileged users on the operating system. As long as the file permissions allow access, such users can easily view the content of the database by browsing the underlying files on the operating system.

3 CONTRIBUTIONS

The crux of our contribution is the design of a holistic database encryption approach which allows organizations to meet their security and compliance requirements without having to make compromises either on the security side or on the database side. Our solution improves over the state of the art discussed above as follows:

- Pervasiveness: All data is encrypted whether it is user tablespace data, system tablespace data, temporary tablespace data, transaction logs data, or database backups data.
- Security: The database content is not vulnerable to attacks by malicious administrators who may choose to bypass the database and access the database indirectly through the file system interfaces.
- Performance: The database system is not forced to dismiss index-based access plans to answer queries with range predicates.
- Breadth: The solution is built into the database engine itself which means that it is available on all platforms where the database system itself runs. Also, it does not force the database system to dismiss the opportunity to bypass the file system and write data directly to raw devices in order to boost performance.
- Quantum-safety: The implementation does not make use of asymmetric encryption to wrap data encryption keys. Data encryption keys are wrapped with symmetric encryption (Chandra et al., 2014). Therefore, the implementation is safe against future attacks by quantum computers implementing Shor's algorithm which is known to break asymmetric encryption that is based on integer factorization such as RSA or on discrete logarithms such as Diffie-Hellman (Shor, 1997). Additionally, the default encryption key size is 256 bits. This also makes the implementation safe against future attacks by quantum computers implementing Grover's algorithm which is known to offer a quadratic improvement in brute-force attacks on symmetric encryption schemes like AES (Grover, 1996).

We have also implemented the solution in a commercial database system (IBM DB2 for Linux, Unix, and Windows).

4 THREAT MODEL

We focus on protecting data at rest. For protecting data in transit between a database server and a client application against eavesdroppers, we assume TLS has been configured to provide this protection. TLS is the standard for protecting data in transit and is implemented by all major database systems.

The content of a database deployed on a given database server can be accessed in two different ways: Directly and indirectly. Direct access is when users interact with the database using the usual database interfaces such as querying the database tables using SQL. In this context, we assume that the database authentication and authorization mechanisms have been configured to ensure that data is accessible only to the appropriate users. Authentication ensures that users are who they claim they are while authorization ensures that authenticated users have access only to those objects or elements within objects for which they have been granted permissions (Rjaibi and Bird, 2004).

Indirect access is when a user chooses to bypass the database system altogether and uses operating system commands to browse the content of the database. For example, on Linux, the following command would display the content of the physical file associated with a given tablespace:

```
strings
'/u01/database/payroll_tbspace'
```

This command will be executed by the operating system bypassing all the database authentication and authorization controls.

Our solution addresses this threat by encrypting the database and ensuring that such encryption is under the control of the database system itself. This means that if a user chooses to bypass the database system as shown above, the operating system command will return cipher text which will be of no value to the attacker.

An attacker may also choose to access the database content from decommissioned hard drives or by physically stealing such hard drives. Our solution addresses this concern as well because the attacker will only find cipher text on those drives. Figure 1 gives a high level overview of our database threat model.

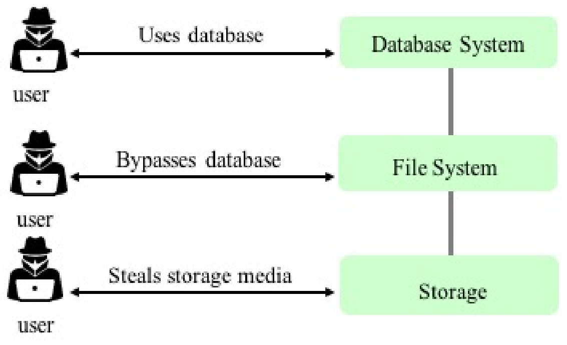

Figure 1: Database threat model.

5 DATABASE ENCRYPTION DESIGN

5.1 Encryption Key Management

Encryption key management is a critical aspect of an encryption solution. Our solution uses two types of encryption keys: A Data Encryption Key (DEK) and a Master Key (MK).

The DEK is the encryption key used to encrypt the actual data in the database. It is automatically generated by the database system at database creation time. The DEK is encrypted with the MK and stored within the database configuration structures together with the following attributes:

- The encryption key size: This is the length of the encryption key in bits (e.g., 256 bits).
- The encryption algorithm: This is the symmetric encryption algorithm used to encrypt the data with the DEK (e.g., AES).
- The master key label: This is the unique identifier of the master key within the external management system. For example, if the external management system is a Hardware Security Module (HSM), then the database system will call out to the HSM and ask it to either encrypt or decrypt the DEK as required. A call to decrypt the DEK is done once when the database system starts up. A call to encrypt the DEK is also done once when the database is created.
- The master key integrity value: To guard against the (rare) event where the MK acquired at some future point in the life of the database is not the one that was actually used to encrypt the DEK, we calculate an integrity value for the MK. We do this by applying a Hash Message Authentication Code (HMAC) function to the MK and store the result. Before making use of the DEK, we first compute an HMAC based on the MK acquired. If the computed HMAC and the stored HMAC match this implies that the master key acquired is indeed the one that was used to encrypt the DEK. Although rare, this is important to avoid corrupting data through decryption with the wrong key.

The MK is the encryption key used to encrypt the DEK. Only a unique identifier of the MK is stored within the database configuration structures. The MK itself is stored in an external key management system such as an HSM.

The reasons for choosing these two types of keys are security, performance, and availability. By storing the MK physically away from the database system, we are assured that compromise of the database system infrastructure does not give the attacker access to both the encrypted data and the encryption keys. Additionally, the concept of MK allows database administrators to rotate encryption keys without impacting the database performance or worse requiring the database to be taken offline to complete the operation. In fact, rotating the MK only requires decrypting the DEK with the old MK and re-encrypting it again with the new MK. In contrast, rotating the DEK requires reading the whole database, decrypting the data with old DEK, re-encrypting it with the new DEK, and writing it back to disk. Thus, the two types of keys we chose in our solution design (DEK and MK) allow administrators to meet their regulatory compliance needs around rotating encryption keys without necessarily having to incur a performance penalty or take a downtime.

5.2 Data Encryption

Implementing security in database systems is always a delicate balance between meeting the security requirements, and ensuring that security coexists in harmony with other critical database features such as performance, compression, and availability. For database encryption, this means that the placement of the encryption run-time processing is key to designing an effective solution.

5.2.1 Encryption Run-time Placement

Our design places the encryption run-time processing just above the database I/O layer in the database kernel stack. The reasons for this choice are the following:

- Pervasiveness: This ensures that all data is encrypted whether it is user tablespace data, system tablespace data, temporary tablespace data, or transaction logs data.

- Transparency: This ensures that encryption has no impact on database schemas and user applications. In fact, encryption can be thought of as invisible to them.
- Performance: This ensures that data stored in the database buffer cache remains in clear text. Consequently, encryption imposes no restrictions on the database system when it comes to selecting the most efficient plan to execute a query (e.g., queries with range predicates).
- Compression: Database systems implement compression techniques to reduce the size of the data stored on disk. Typically, these techniques look for repeating patterns in order to avoid storing all copies of such patterns. Encryption, by definition, removes all patterns. This means that the order in which compression and encryption are performed is important. For example, if encryption is performed first, then the compression rate will be zero as encryption will leave no patterns. Thus, placing our encryption run-time processing just above the database I/O layer ensures that encryption and compression can coexist in harmony.

5.2.2 Encryption Run-time Processing

The encryption run-time processing consists of two functions: Encryption and decryption. Encryption takes place when the database system is writing data out to the storage system. Decryption happens when the database system is reading data in from the storage system.

While the solution can easily support any symmetric block cipher for encryption/decryption, we have chosen to implement support for only AES and 3DES as they are the most commonly used block ciphers. AES is actually the standard symmetric block cipher. Block ciphers support many modes of operations. Electronic Code Book (ECB) is the easiest mode to implement but is also the weakest from a security perspective. This is because in ECB mode the same clear text input will always result in the same cipher text. This may be fine for encrypting small pieces of data such as a password, but not for database encryption as this will introduce patterns and may compromise the encryption solution. Instead, we have chosen to use the Cipher Block Chaining (CBC) mode as it does not introduce patterns. This means we need to provide an initialization vector when calling the block cipher in CBC mode for encryption, as well as maintain that initialization vector in our meta-data so that it is available for decryption purposes. Note that the initialization vector is not meant to be a secret. It only needs to be random.

When writing data to the file system, the database system writes them in chunks to minimize the I/O overhead. A chunk is a collection of data pages where each page is 4KB in size. A page is set of rows, and a database table is a collection of pages. This poses an interesting question as to the level of granularity to adopt for encryption. We have chosen the data page to be that level granularity. A row level granularity would have had a higher impact on performance as encryption calls would have to be made for each row separately. A chunk level granularity would have created a dependency between the pages in that chunk due to the chaining inherent to the CBC mode. For example, to decrypt page 5, one must first decrypt pages 1, 2, 3, and 4. This would have had a negative impact on query performance as it diminishes the value of index-based access.

It is also worth noting that the data page level granularity has allowed us to avoid having to needlessly increase the database size due to encryption. In fact, encryption block ciphers such as AES and 3DES encrypt data one block at a time. For example, the block size for AES is 16 Bytes. This means that when the clear text to encrypt is not an exact multiple of the block size, padding is required and this obviously increases the cipher text compared to the original clear text. Fortunately, the choice of a data page for the encryption granularity avoids this problem as data pages are always an exact multiple of the encryption block size.

5.2.3 Transaction Logs

Transaction logs are files where the database system logs transactions such as insert, delete, and update operations. They are a critical component for ensuring the integrity of the database as well as for allowing recoverability of the database following a database crash. The structure of a transaction log file consists of two pieces: A header which contains meta-data about the file, and a payload which contains the actual database transaction details.

In section 5.2.2 above, we have seen how the placement of the encryption run-time ensures that all data written to disk, including transaction logs, is automatically encrypted. However, transaction logs pose one additional challenge. In a database recovery scenario, we must be able to decrypt the transaction logs even when the database system is down. This means that we cannot rely on the DEK related information (section 5.1 above) to decrypt the transaction logs as the database system may be offline. To address this challenge, the transaction logs structure has been extended so that these logs are self-contained when decryption is required. More specifically, the header piece of the transaction logs structure has been extended so that it contains its own copy of the DEK related information. This also opens the door for an opportunity to further boost security by generating a separate DEK for the transaction logs that is distinct from the DEK for the database.

5.2.4 Database Backups

A database backup is a copy of the database content at a given point in time. Database systems provide a command and/or API to allow users to take those backups. In the case of a database crash, the database can be recovered to the state it was at when the last backup was taken. Additionally, when healthy transaction logs from the damaged database are available, it is possible to recover the database to a further point in time by reapplying the database transactions from the transaction logs. Like transaction logs, a database backup consists of two pieces: A header which contains meta-data about the backup, and a payload which contains the actual copy of the database.

Database backups pose the same challenge as transaction logs in the sense that they too need to be self-contained when decryption is required. Consequently, this challenge is addressed in the same way by extending the database backup header piece so that it contains its own copy of the DEK related information. Like transaction logs, database backups have their own unique DEK.

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented a holistic approach to database encryption which allows organizations to meet their security and compliance needs without having to make compromises either on the security side or on the database side. Figure 2 gives a high level overview of the architecture, which we implemented in IBM DB2 for Linux, Unix, and Windows.

In our future work, we intend to enhance our holistic database encryption solution to better address two challenges. The first challenge is encrypting existing databases. Unlike newly created databases, an existing database already has data and turning encryption on for that database means not only encrypting new incoming data, but also encrypting that existing data. The current solution requires the organization to turn on the encryption for the existing database during a scheduled database maintenance window. This is because the current approach for encrypting an existing database works by having the database administrator take a backup of the existing database and then restoring it using the RESTORE DATABASE command. While processing the restore, the database system encrypts the data as that is analogous to new incoming data. We would like to allow database administrators to turn on encryption for their existing databases without having to wait for a scheduled maintenance window. To do so, we plan to investigate creating a background process which encrypts the database incrementally while the database system continues to serve applications. The main challenge would be finding out how to perform this incremental encryption without compromising the data integrity.

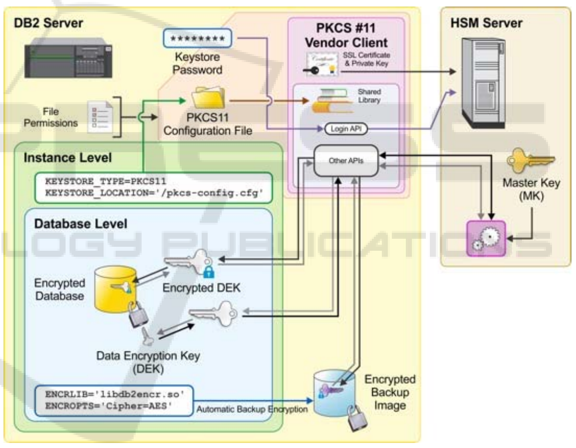

Figure 2: Database encryption architecture.

The second challenge is rotating the DEK online. Currently, our solution allows rotating only the MK online. While rotating the MK is usually sufficient, there may be situations where rotating the DEK itself is required. Currently, the only way to do this is during a scheduled maintenance window following the same database backup and restore discussed above. We believe that the solution for encrypting existing databases without having to wait for scheduled maintenance window would also allow rotating the DEK online as that is fundamentally the same problem. That is, in both cases, the database content needs to be read, re-encrypted with a new DEK, and written back to disk.

ACKNOWLEDGEMENTS

The author would like to thank Saifedine Rjaibi and Devan Shah for their valuable comments.

REFERENCES

Rjaibi, W., Bird, P., 2004. A Multi-Purpose Implementation of Mandatory Access Control in Relational Database Management Systems. In *VLDB'04, 30th International Conference on Very Large Data Bases*. Morgan Kaufmann.

Chandra, S., Paira, S., Alam, S., Sanyal, G., 2014. A Comparative Survey of Symmetric and Asymmetric Key Cryptography. In *ICECCE'14, International Conference on Electronics, Communication and Computational Engineering.* IEEE.

Grover, L., 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *STOC'96, 28th Annual ACM Symposium on Theory of computing*. ACM.

Shor, P., 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing, Volume 26 Issue 5.

Dufrasne, B., Brunson, S., Reinhart, A., Tondini, R., Wolf, R., 2016. *IBM DS8880 Data-at-rest Encryption*, IBM Redbooks. New York, 7th edition.

Benfield, B., Swagerman, R., 2001. Encrypting Data Values in DB2 Universal Database. IBM DeveloperWorks.

Anto, J., 2008. Understanding EFS. IBM DeveloperWorks.

Freeman, R., 2008. *Oracle Database 11g New Features*, McGraw-Hill.

Voigt, P., Von Dem Bussche, A., 2017. *The EU General Data Protection Regulation (GDPR)*, Springer International.

Chuvakin, A., Williams, B., 2009. *PCI Compliance: Understand and Implement Effective PCI Data Security Standard Compliance*, Elsevier.