Understanding and Selecting a Database Encryption or Tokenization Solution

Author's Note

The content in this report was developed independently of any sponsors. It is based on material originally posted on the Securosis blog but has been enhanced, reviewed, and professionally edited.

Special thanks to Chris Pepper for editing and content support.

Licensed by

This paper is not currently sponsored or licensed.

Contributors

The following individuals contributed significantly to this report through comments on the Securosis blog and follow-on review and conversations (in no particular order):

Arshad Noor

Boaz Gelbord

Clarkendweller

Dave Howe

JohnF

Mostafa Siraj

pktsniffer

Sharon Besser

Terence Spies

Copyright

Table of Contents

Introduction

Over the last two decades, database security has meant access controls and encryption. Access controls to gate who should and should not be allowed access to the database, and encryption to protect data at rest. The use of access control systems for databases is well documented, and the available solutions are very effective at providing basic security around who can access what data. Somewhat surprisingly, database encryption is still not in wide use. Data security professionals often view database encryption as a redundant control, only effective in the event other security measures and polices fail. Application developers have avoided database encryption because the burden of implementation would land on them, adding complexity to the design and implementation of both application and database schemas. IT managers as a rule dislike the additional complexity around backup, recovery, user provisioning, and key management. And everyone is wary of the specter of performance degradation. While there is validity to each of these perspectives, they are relevant only in particular circumstances. Regardless, these perceptions relegate database encryption to a secondary measure, to be considered as part of a 'defense in depth' strategy. As we all know, 'nice-to-have' security measures remain academic.

Two fundamental changes have occurred which alter perceptions of database encryption. First, several public data breaches could have been prevented or neutralized had the compromised databases been encrypted. In response, government and private regulations, such as the PCI Data Security Standard, have either endorsed or mandated encryption as a security measure to reduce data theft. Second, database and IT product vendors have dramatically improved the performance and manageability of their encryption offerings. Reduced implementation and,management burdens, coupled with (in many cases) reduced cost of encryption have reduced internal resistance. Still, encryption's ability to solve business security concerns can be complicated to fully grasp, and the fear of building a system that conceals management nightmares which only becomes visible in production, is widespread.

This paper begins with a decision tree to help you determine which kind of database encryption you need based on your security and manageability requirements. Next we describe the different database encryption options — showing how each variant affects application performance, manageability, and security. We'll cover both traditional encryption and the newer alternatives: tokenization and Format Preserving Encryption. We also review key management options, the role of access controls, and application-level encryption. Finally, we close with a real-world example of how to select a database encryption or tokenization solution to meet your organization's security objectives. We feel the best way to accomplish these goals is to present a simple guide for selecting a database encryption strategy — basically a decision tree for you to plug in your requirements and map those to the available database security choices. This process will highlight the differences in approaches, and tie business needs to technical capabilities. Between this guide and the use cases, end users will have the tools to both clarify their goals and determine an appropriate implementation strategy.

Types of Database Encryption

Before we discuss which problems database encryption helps mitigate, we have to define exactly what it is. The term *database encryption* is used to describe many different methods of data protection, implemented either outside or within the database engine. Conceptually, a database is a sophisticated box to put data in. Taking this analogy one step further, you can protect the entire box (File/OS), the entire contents of the box (Full database), or some subset of the content within the box (Column, Table, Schema). We can apply encryption to the contents through native database functions or externally with third party tools, but both are called database encryption. We also see growing use of *tokenization* as an alternative or complement to encryption, since it achieves many of the same goals, which is why we include it in this paper.

For this discussion, we divide database encryption into two basic types:

Transparent/External Encryption:

These terms refer to encryption of the entire database. This is provided by native encryption functions within the database engine. Some database vendors offer column and table level granularity, but it is increasingly common to apply encryption to all the data. We call this 'transparent' database encryption because it is invisible to the applications and users that use the data, and requires no changes to application logic. The principal use case is to prevent exposure of information due to loss of the physical media (disk, tape, etc.) or compromise of the database files in storage. Transparent encryption can also be handled through drive or OS/file system encryption, applying encryption on everything that gets written to disk. Although these options may lack some of the protections of native database encryption, both are invisible to the application and do not require alterations to the code or schemas. Transparent encryption protects the database from users without database credentials, but does not protect data from authorized users.

User/Data Encryption:

These terms describe encrypting specific columns, tables, or even data elements within the database. We call this 'user' encryption because the objects being encrypted are owned and managed on a per-user basis. Tokenization also falls into this category. The classic use case for this encryption model is encrypting credit card numbers within a database. The goal is to provide protection against inadvertent disclosure, or to enforce separation of duties on credentialed users of the database. The downside is that these variants are not invisible to the application and usually require code and database changes. The concept is to encrypt only the highly sensitive data we are worried about, reducing the overall performance impact, and minimizing code and database changes. How this is accomplished depends on how key management is handled, the use of internal vs. external encryption services, and how applications use the database.

It should be stressed that *both* user and transparent encryption protect media, but the granularity provided by user encryption comes at the cost of required modifications to code and/or database schemas. Some vendors offer transparent encryption applied to specific tables or columns, but the value proposition is still focused on lost media and file protection, not separation of duties.

Tokenization or Format Preserving Encryption are designed to reduce required external application changes, but still require internal database modifications.

In some of our earlier essays and articles, we described these two options as "Encryption for Media Protection" and "Encryption for Separation of Duties". That was a bit of an oversimplification, but those terms better describe the value they provide vs. the technology features that do the work. While we wanted to use terminology that directly mapped to the business use case, if you asked your RDBMS vendor for "media encryption", they would be unlikely to know what the heck you were talking about.

In summary, transparent/external encryption protects data from compromise by attacks from outside the database, but does not protect against credentialed database users. User/data encryption can encrypt data and restrict access based on database users, but at a higher cost in terms of complexity, performance, and manageability.

The Selection Process

During the selection process for database encryption solutions, too often the discussion quickly devolves into encryption technologies: algorithms, computational complexity, key lengths, merits of public vs. private key cryptography, key management, and the like.

In the big picture, none of these topics matter.

While these nuances are worth considering, a focus on technologies misses the primary business driver of the entire effort: what threat do you want to protect the data from? If you are going to encrypt database contents, there must be a compelling reason. We think it's safe to say that if you are looking at database encryption as an option, you have already come to the decision that you need to protect your data in some way. At the very least you will incur a performance penalty, and in the worst case significant alterations to the application and database infrastructure. There must be a compelling business driver to justify the time, expense, and performance impact.

What is the Business Need?

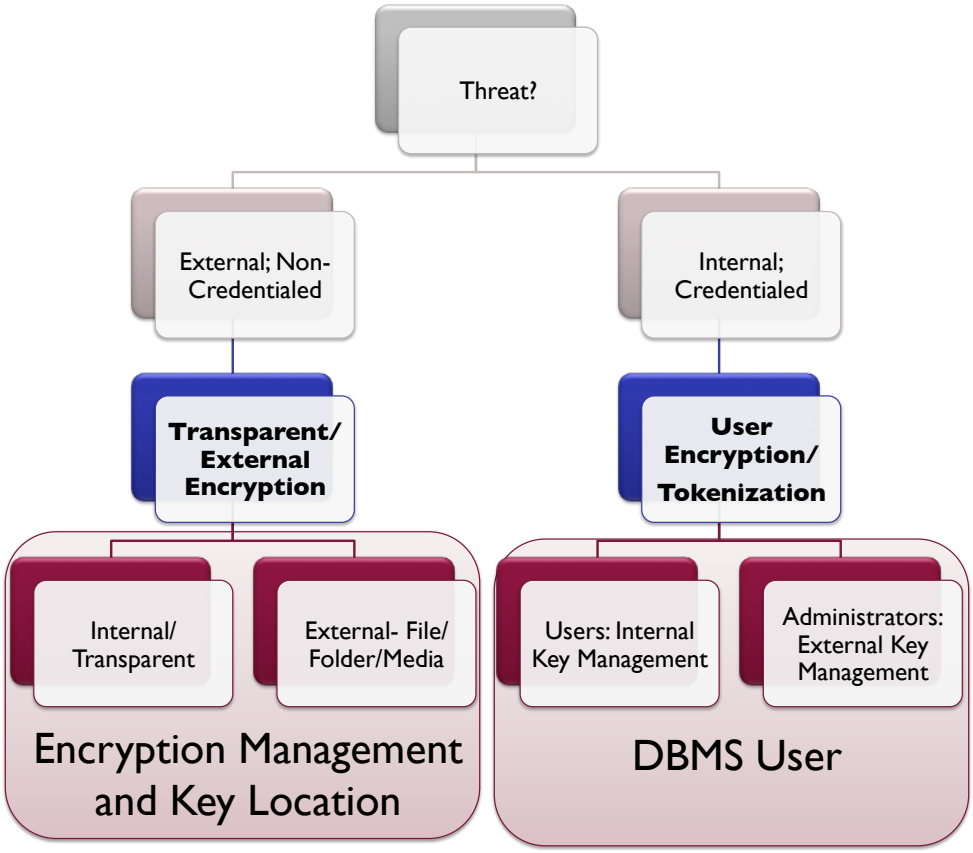
The first thing to decide when looking at database encryption is what are we trying to protect and why. If we're just going after the 'PCI checkbox' or worried about losing data from swapping out hard drives, someone stealing the files off the server, or misplaced backup tapes, then transparent encryption is your answer. Blanket encryption of all database content for media protection is much easier than encrypting specific columns and tables for separation of duties. If the goal is to protect data in the event of compromised accounts, rogue DBAs, or inadvertent disclosure things get a lot more complicated. We will go into the details of *why* and *how* as we delve into the specific implementation issues with user encryption later in this paper. For now, which path is right for you depends upon three things:

- What do you want to protect?
- What do you want to protect it from?
- What application changes and management tasks can you tolerate?

You need a clear, *documented* business need before progressing any further with this exercise. Make sure you understand both the threat and the data to protect. If you are adopting encryption for a regulatory requirement, speak with an auditor so you are fully aware of the options and requirements prior to moving forward.

The Decision Tree

Once you've identified the business problem, we can map that requirement to the available technologies to achieve the goal. With your requirements in hand, let's walk through the following diagram to understand how the decision process works:



Whether your primary driver is security or compliance, the breakdown will be the same. Whether you need to provide separation of duties for Sarbanes-Oxley, or protect against account hijacking, or keep credit card data from being viewed for PCI compliance, you are worried about credentialed users. In this case you need a more granular approach to encryption and possibly external key management. We call this User Encryption. If you are worried about missing tapes, physical server theft, copying/theft of the database files via storage compromise, or un-scrubbed hard drives being sold on eBay, the threat is outside the bounds of access control. For these cases transparent/external encryption through native database methods, OS support, file/folder encryption, or possibly drive encryption is appropriate.

Once you have decided which method is suitable, we need to examine the basic technology variables that affect your database system and operations. Your selection controls how much of an impact it will have on applications, database performance, and so on. With any form of database encryption there are many technology variables to consider for your deployment, but for determining the right ***strategy***, there are only three to worry about. These three affect performance and which types of threats you can address. In each case we need to determine whether these operations will be performed internally by the database, or externally. They are:

1. Where does the encryption engine reside? [inside/outside]
2. Where is key management performed? [inside/outside]
3. What performs the encryption operations? [inside/outside]

Each option moved outside the database means more complexity and less application transparency, but (generally) stronger security through increased separation of duties. For example, with basic transparent encryption the encryption engine (the code that performs the encryption), key management, and encryption operations management are all within the database. The database administrator has full control of encryption operations, and in many cases access to the keys as well. A more secure option might be to remove key management to an external application, so the DBA is never responsible for the keys, which improves separation of duties.

Throughout the remainder of this paper, we will discuss the major branches of this tree, and how they map to threats. We will follow up with a set of use cases to contrast the models and set realistic expectations for security they can and cannot provide, as well as some comments on the operational impacts of these technologies.

By the end you'll be able to walk through our decision tree and pick the best encryption option based on what threat you're trying to manage, along with operational criteria ranging from what database platform you're on to management requirements.

Transparent/External Encryption

In this section we dig into the first branch of the decision tree from the previous section: non-credentialed threats. To recap, we are discussing threats against the data that originate from outside the database using non-database tools. Both options (transparent and user) protect data on storage media, but transparent encryption is far easier to use. But if you need to protect data, such as credit card numbers, from even database administrators, this option is insufficient. If you are primarily worried about lost media, stolen files, a compromised host platform, or insecure storage, then Transparent Encryption is your best choice. Our goal here is to understand the threats and how transparent encryption meets the challenges. That sounds simple enough, but it can be tough to wade through vendor claims, especially when everyone from network storage to database vendors claims to provide the exact same value. We need to understand how to deal with the threats conceptually before jumping into the more complex technical and operational issues that complicate your choices.

If you have a database you already have access controls to protect its data from unauthorized access through normal database communications. The database itself screens queries or applications to make sure that only authenticated users and groups are permitted to examine and use data. However, there are many ways for database data to be accessed without credentials being supplied at all. The threat to address here is protecting data from physical loss or theft (including some forms of virtual theft) through means that are outside the scope of access controls. Keep in mind that even though the data is 'in' a database, that database maintains permanent records on disk drives, and data is archived to many different types of long-term storage.

If the primary risk to address is complete bypass of the database engine — for example, examination of backup tapes, disks, offline redo log files, transaction logs, or in any other form on storage media — transparent encryption is appropriate. Anything from file editors to debugging tools can load and view the raw contents of the database (even when the archive is in binary format) and translate into ASCII if necessary, and re-assembled to recreate the original records. Programers can reverse engineer the data structures used by database vendors to recreate data on a table-by-table, row-by-row basis. This takes a bit more time than writing a SQL query, but is no less effective for collecting database contents.

Encrypting data on media is an excellent way to protect it from unwanted inspection. Further, as the media is usually kept separate from the encryption keys, there is very little room for any option accept a brute force attack on the encrypted media. We call this *Transparent Encryption*, as do many database vendors, to describe the capability to encrypt data stored inside the database without modification to applications using it. We add 'External' to distinguish between internally managed database features and encryption provided at the file, OS, or media level. Let's examine a few of the available options in greater detail:

Transparent Encryption Options

Transparent/External Encryption for protecting database data uses the following techniques & technologies:

- Native Database Object (Transparent) Encryption: Database management systems such as Oracle, Sybase, Microsoft SQL Server, and IBM DB2 include capabilities to encrypt either internal database objects (tables and other structures) or the data stores (files). These encryption operations are managed from within the database using native functions, with keys stored in database tables by default. This is a good overall option for protecting data on media as long as performance meets your requirements. It is by far the easiest to implement, and given that it is entirely self-contained, requires no modification to existing operations or IT infrastructure. For most customers there are additional licensing costs to use these features. Depending on the platform, you may be able to offload key management duties to an external key management solution for added security and reliability. External key management has the disadvantages of added cost, increased complexity, and limited support from some platforms.
- External File/Folder Encryption: The database files are encrypted using an external (third party) file/folder encryption tool. Assuming the encryption is configured to encompass all of the necessary data files, this protects the database files from unauthorized access on the server and those files are typically still protected as they are backed up, copied, or moved. Keys should be stored off the server, and no access provided to local accounts, which protects against the server being compromised by an external attacker. Some file encryption tools, such as Vormetric and BitArmor, can also restrict access to the protected files by application. Thus only the database processes can access the file, and even if an attacker compromises the database's user account, they will only be able to access the decrypted data through the database itself. File/folder encryption of database files is a good option as long as performance is acceptable and keys can be managed externally. Any file/folder encryption tool (including Microsoft EFS) should support external key management, but performance must to be tested because there is wide variation across tools. Remember that any replication or distribution of data handled from within the database won't be protected unless you also encrypt the destinations. This could mean that your tape archives are not encrypted, so check with the vendor. Also remember that some offerings affect database restarts, so you need to understand how key access is managed to support your database operations. External encryption products provide the added benefit of encrypting non-database files and folders as well.
- Media encryption: This includes full drive or SAN encryption; the entire storage media set is encrypted, and thus the database files are protected. This is best suited for high performance databases where the primary concern is physical loss of disk drives (*e.g.,* a database on a managed SAN where a service provider handles failed drives potentially containing sensitive data). These solutions also work with external key servers for better security and disaster recovery. Depending on the method and specifics of the environment, this may or may not provide protection for the data as it moves to other data stores, such as archival (tape) storage. For example, depending on your backup agent, you may be backing up the unencrypted files or the encrypted storage blocks, so investigate what your vendor offers. Media encryption is well suited to a handful of specific use cases, so careful consideration is needed.

You will notice that, depending on which technique you choose, the initiation of the encryption, the engine that performs the encryption, and the key management server may all reside in different places. In fact, the latter two techniques encrypt the database but take place entirely outside it — the engine that performs the encryption, as well as the processes responsible for managing encryption operations, are external. In all three cases encryption remains transparent to business processing functions. You have many options, and which to choose depends on your performance requirements, threat model, cost, existing architecture, and security requirements. In most cases it's preferable to use external key management and not allow access from any local accounts. As external key management has added cost, many customers we speak with feel the benefit is not worth the added expense, and choose to mitigate risk through segregation of administrative duties. For database systems not used for heavy transaction processing,

current versions of relational database platforms offer reasonable performance and seamless integration cost effectively. We will outline selection criteria and use cases to support the decision process later in this document.

User Based Encryption and Tokenization

In this section we detail the other half of the decision tree for selecting a database encryption strategy: securing data from credentialed database users. Specifically, we are concerned with preventing misuse of data, through individual or group accounts that provide access to data either directly or through another application. For this paper we are most interested in differentiating between user-level and administrative accounts. These are the two primary types of credentialed database users, and each needs to be handled differently because their access to database functions is radically different. As administrative accounts have far more capabilities and tools at their disposal, the threats they pose are more varied and complex, making it much more difficult to safeguard sensitive data. Also keep in mind that a 'user' in the context of database accounts may be a single person, a group account associated with multiple users, or an account utilized by a service or program.

We call this user encryption (and tokenization) because we are assigning access (keys) on a per-user basis. This differs from transparent encryption in two important ways. First, we are now protecting data accessed through the normal database communication protocols, as opposed to blocking inspection that bypasses the database engine. Second, we are no longer encrypting everything in the database — instead we want to encrypt as little as possible so non-sensitive information remains available to the rest of the database community. Conceptually this is very similar to the functionality provided by database groups, roles, and user authorization features. In practice it provides an additional layer of security and authentication where, in the event of a mistake or account compromise, exposed data remains encrypted and unreadable. Since regular users can be effectively restricted using access controls, *encryption at this level is mostly used to restrict administrative users*.

Let's take a closer look at the threats we want to address, and then examine our encryption options more closely.

User Encryption Options

They say when all you have is a hammer, everything begins to look like a nail. That statement is relevant to this discussion of database encryption because most database encryption vendors begin the conversation with their capabilities for column, table, row, and even cell level encryption. But these are simply tools, and for what we want to accomplish they may even be the wrong tools. We need to fully understand the threat first (in this case credentialed users) and build our protection model based on that and what we're trying to protect. For credentialed user threat analysis, we assume that something will go wrong, allowing someone to leverage credentials to access sensitive information which would normally

be blocked by access controls or account limitations. Otherwise, the access controls would be enough and there would be no need for encryption at all.

Previously we asked the questions "What do you want to protect?" and "What threat do you want to protect the data from?" It might make more sense to ask "*Who* do you want to protect the data from?" General users of the data or administrators of the system? Let's look at the risks associated with these two groups in detail:

- Users: This is the general class of users who call upon the database to store, retrieve, report, and analyze data. They may do this directly through queries, but far more likely they connect to the database through another application. There are several common threats companies typically want to address for this class of user: providing protection against inadvertent disclosure from sloppy privilege management, inherited trust relationships, meeting a basic compliance requirement for encrypting sensitive data, or even providing finer-grained access control than is otherwise available through the application or database engine. Applications commonly use service accounts to connect to the database; those accounts are shared by multiple users, so the permissions may not be sufficiently granular to protect sensitive data. Users do not have the same privileges and access to the underlying infrastructure that administrators do, so the threat is exploitation of lax access controls. To protect against this, we need to identify the sensitive information, determine who may use it, and encrypt it so only appropriate users have access. In these cases deployment options are flexible, as you can choose key management that is internal or external to the database, leverage the internal database encryption engine, and gain some latitude as to how much of the encryption and authentication is performed outside the database. Keep in mind that ***access controls are highly effective with much less performance impact, and they should be your first choice***. Only encrypt when access controls aren't sufficient and encryption really buys you additional security. One possibility is to place the encryption engine outside the database, and only decrypt content for a specified service account on an application server. Thus if the service account is compromised within the database via direct access, the content is still protected. As you can see, this is a rather specialized scenario.
- Administrators: The most common concern we hear companies describe is their desire to mitigate damage in the event that a database administrator (DBA) account is compromised or misused by an employee. This is the single most difficult database security challenge to solve. The DBA role has rights to perform just about every function in the database, but no legitimate need to examine or use most of the data stored there. For example, during normal operation the DBA has no need to examine Social Security Numbers, credit card data, or any customer data to maintain the database itself. When the requirement is to protect the data from highly privileged administrators, enforcing separation of duties and providing a last line of defense for breached DBA accounts, then at the very least external key management is required. By encrypting and removing key management from DBA control, sensitive information can be kept secure. External key management provides separation of duties between management of the database and use of the data therein. Orchestration of the encryption/decryption functions is typically performed by the application rather than the database, requiring modification of the application code. Use of the database engine's built-in encryption capabilities may be possible, depending on the vendor implementation, and most vendors provide APIs for third party encryption support while maintaining a single database 'conversation'. This design addresses user-level threats as well, and should be considered a superset. Once again, if the database offers separation of duties of the administrative role, investigate that first, as it is considerably easier to implement and fairly effective at reducing risk. Remember that the encryption only protects against unauthorized DBA access if the administrator is restricted from the data in the first place (the account isn't authorized to decrypt the data) and you have additional controls, such as activity monitoring, to limit the ability of DBAs to hijack authorized user accounts. This is why encrypting the data outside the database is often attractive.

Once we've decided which of these threats to address we can select tools, technologies, and deployment options to accomplish our goals. We have established that at the very least, the two drivers will indicate either internal or external key management. Depending upon your answers to the question "What data do you want to protect?", we can now decide at what level to encrypt (tables, columns, rows, or cells), the type of key management needed, whether to leverage the internal database encryption engine or use external services, whether tokenization is an option, and what changes need to be made to the business logic. We cover these topics in the next section, and follow up with several common use cases.

With user encryption, we assign access rights to the data we want secured on a per-user basis, and provide decryption keys only to the specified users with authorization for that information. While keys can be automatically supplied to users when they log into the database, as with transparent encryption, a secondary authentication and authorization process for key access is recommended to safeguard data in case accounts are hijacked. In this way the keys are not tied to the user account, and data is not automatically vulnerable should a user's password be guessed. With tokenization, the data is replaced be a representative or *token* value that is only linked to the sensitive data in a secure repository, where the data is also typically encrypted.

Access Controls vs. Encryption

One very common question that comes up when evaluating user encryption is, "How is this different than access controls?" The distinction is in how you use them. Encryption's value is in providing a level of granularity beyond what's possible with access controls, protecting data as it moves (physically or virtually), and robustness. Think of it as a lockbox — you can use it to protect something in the mail, or in a secure room to keep it safe from the guard at the door. In a practical sense this means restricting administrators, who have access to keys inside (DBA) or outside (IT Admin) the database, since access controls are very effective for all other users. Another more complex option is to use digital certificates outside the database, adding (essentially) another authentication factor. This increases security, because simply compromising a username and password isn't sufficient to read the data, and so is particularly useful for protecting data utilized by service accounts.

For the most part, as you'll see in our use cases, we only recommend user-level database encryption under extremely limited circumstances. It's a complex topic, and we haven't even dug into the technology yet, but please don't assume because we are spending so much time on it that it's your best option. *Just because it's complicated and takes a long time to describe, doesn't mean that's what you need to achieve data security.*

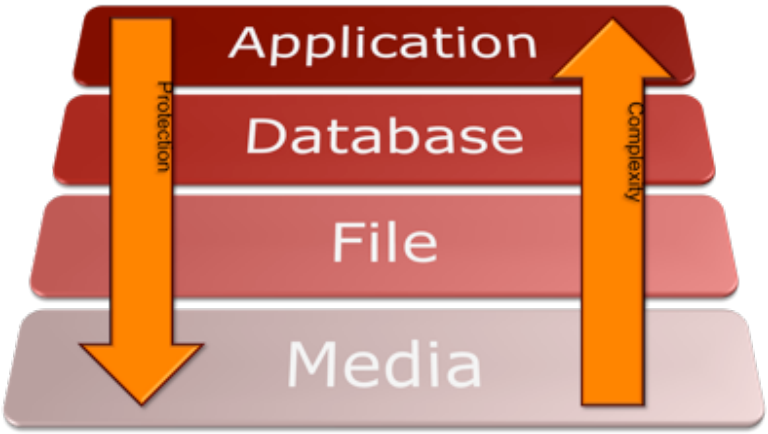
Database vs. Application Encryption

There is a lot of confusion around the differences between user encryption with external key management and application encryption. What's the difference? Both require that the application initiate much of the activity, require application and database alterations, and manage keys outside the database. The distinction is where the encryption engine resides: inside or outside the database. The application will either make procedure calls to a library outside of the database, or procedures calls to a library embedded within the database.

The following diagram describes potential encryption layers. What we find is that the higher up the stack you encrypt, the greater the overall protection (since data stays encrypted through the rest of the layers), but this increases complexity. It's far easier to encrypt an entire hard drive than a single field in an application; at least in real world implementations. By

giving up granularity, you gain simplicity. For example, to encrypt the drive you don't have to worry about access controls, tying in database or application users, and so on.

In an ideal world, encrypting sensitive data at the application layer would generally be the best choice. Practically speaking, it's not always possible, as your application or database architecture might not support it. It's very hard to design appropriate application level encryption, even when you're using crypto libraries and other adjuncts like external key management. Database encryption is also very difficult to get right, but is sometimes slightly more practical than application encryption. When you have a complex, multi-tiered application with batch jobs, OLTP connections, and other components, it may be easier to encrypt at the database level and manage access based on user accounts (including service accounts). Once again, this is why we call the model "user encryption".



Keep in mind that if someone compromises user access controls, keys are automatically provided, and encryption is worthless. In the event this happens, additional controls like application-level logic or database activity monitoring might be able to mitigate a portion of the risk, but generally once your account is compromised your data is exposed. For retail transactions, especially those under the auspices of PCI, application encryption (done properly) or tokenization are preferred. They provide redundancy in user authentication and additional controls over data access. For many companies we work with, they are not viable in the short term, given the difficulty of implementation. Instead they choose to start with some sort of database encryption (usually transparent/external) to address compliance and risk requirements immediately. Once again application encryption isn't a panacea — it can work well, but brings additional complexities and is very easy to screw up. Use with caution.

Tokenization

With the increasing focus on payment transaction security due to the quantum-entangled forces of breaches and PCI, we are seeing a revitalization of tokenization as a security control, particularly when handling credit card numbers or other semi-structured data. Tokenization technology works by swapping a value such as a credit card number (or other data, but cards are the most common usage today), with a *token*. Rather than trying to encrypt all credit card data in all merchant and payment processor databases, the strategy is to keep the numbers in one master database, and then replace the card numbers with *tokens* in all other systems. The master database can be hardened and encrypted, and keeps track of which token matches which credit card. Other systems send the tokens to the master system for processing, which then interfaces with external transaction processing systems. By swapping out all the card numbers, most of the data security effort can be focused on one system that's easier to control.

Tokenization need not use encryption, but the two are closely linked. In nearly every commercial example we've seen, the original credit card or PAN data is encrypted in the master database, with the original data providing the link to the token. Here we will describe implementing tokenization for payment processing, including point of sale terminals, but we've seen similar deployments for web-based payments, to protect other private data such as Social Security Numbers at academic institutions that previously used them as student identification numbers, or in healthcare applications. Keep in mind this is an overview of tokenization as it applies to databases, and not a complete review of tokenization solutions.

Tokenization systems create a proxy for sensitive data, with each token filling in as a reference for the original. Tokens can be the same size as the original data, and may even follow the same format (such as a token for a credit card number being random, yet passing a LUHN check). As the token still resembles a card number (perhaps even including the same last 4 digits, while the rest of the data is randomized), systems that use card numbers do not need to be altered to accommodate tokens. Unlike the original credit card number, tokens cannot be used as financial instruments, and have no value other than as a reference to the original transaction or the real account. The relationship between a token and a card number is unique for any given payment system. Thus even if someone compromises the entire token database in a way that still allows them to commit transactions in that system (a rare possibility), those numbers are worthless to the outside world. And most important, the token cannot be decrypted into the original credit card number. These same principles can apply to any type of data with a standardized format, or tokens can be completely random with no pattern if the application doesn't require it.

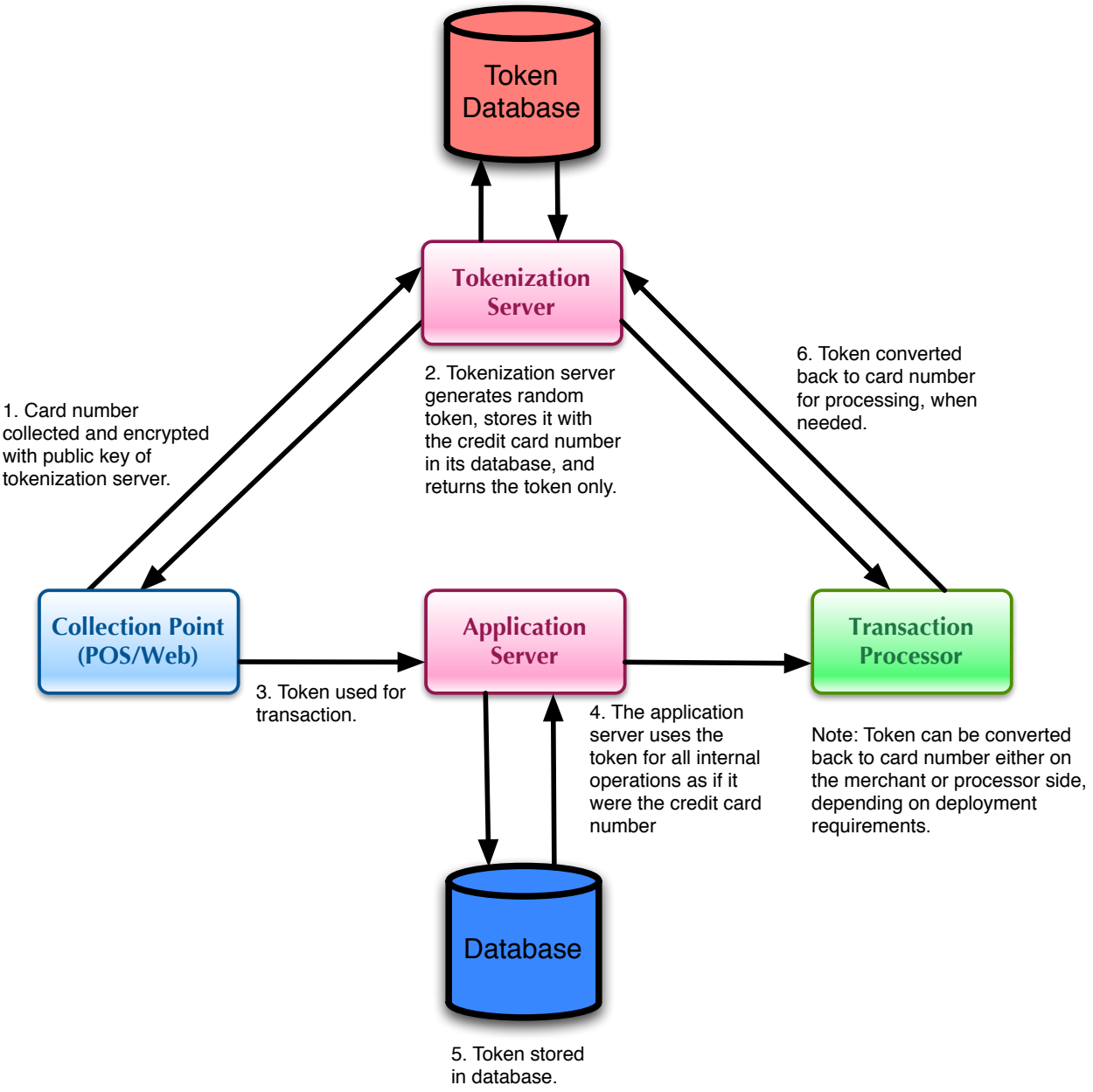
The credit card is still encrypted at the point of sale (POS), and passed in an encrypted state back to the payment processor. The processor creates a token that represent the credit card number, which is then passed back to the merchant. The token is the stored value on the merchant site, and since it's paired with the real number on the processor side, it can still be used for refunds and other transactions. Some implementations always require the original card for new purchases, but only the token for anything else.

Thus the real card number is never stored in the clear (or even encrypted) on the merchant side. There's really nothing to steal, which eliminates any possibility of a card number breach (according to the Data Breach Triangle). The processor is still at risk, so they need to use different techniques to lock down and encrypt the actual card numbers. If processing services are distributed across several sites, encryption keys must be distributed and risk to the data remains higher. Encryption is only a stopgap in most organizations, and once you hit the point where you have to start making application changes anyway, go with tokenization where possible. Even payment processors should be able to expand use of tokenization, relying on encryption to cover the (few) databases which still need the PAN (Primary Account Number).

Organizations can also implement tokenization completely within their own environments. In these cases the tokenization server resides internally, but in a single, heavily secured database with extremely limited access. This allows business units to manage their own systems, without the risk of exposing protected information. The tokenization server/database is the only point of contact with any payment transaction system, reducing overall risk and compliance scope. This model also works for non-payment systems where a tokenized value reduces security risk, such as a university which needs to replace Social Security Numbers historically used as student numbers. In such an environment, most systems might simply use the tokens as ID numbers without ever needing to look up the actual SSNs.

As you can see, this model works in a variety of circumstances with widely distributed data in a particular format that must be protected, without requiring radical changes to the sensitive data fields which would break application compatibility. Tokenization is gaining popularity because it substantially improves security and constrains compliance scope, without imposing widespread and major application changes. Payment processors love tokenization, because it nearly eliminates wide-scale breaches on the merchant side.

Generic Example of Tokenization for Credit Card Processing.



Format Preserving Encryption

Format Preserving Encryption (FPE) is not a new concept, but has lately been getting much more attention due its inherent advantages for dealing with the PCI requirement to encrypt credit cards. With many traditional ciphers, encrypting a small number returns a larger hex or binary value. Because of the larger size and different data type, database applications require alterations to database structure and associated queries to accommodate the larger ciphertext. FPE is also called Datatype Preserving Encryption (DPE) and Feistel Finite Set Encryption Mode (FFSEM), among other names. Technically there are many labels to describe subtle variations in the methods employed, but collectively they attempt to retain the same size — and in some cases the data type — as the original data being

encrypted. For example, encrypt “408-555-1212” and you might get back “192807373261” or “a+3BEJbeKL7C”. The goal is to substitute encrypted data without needing to change systems that use that data — such as database structures, queries, and application logic.

While it may be surprising, some variants offer encryption as strong as AES (Advanced Encryption Standard). Specifically, FFSEM is a variant of AES, and has been mathematically proven no less secure than classic AES.

The business justification for this type of encryption is typically to encrypt the data and minimize application changes, when complete centralization via tokenization isn’t viable. FPE is typically used when changing a) the database or other storage structures, or b) the applications that process sensitive data, would be prohibitively complex or expensive. FPE can a way to protect the data without requiring such intrusive changes. Keep in mind, though, that even if all the existing application code can remain unchanged, you will still need to modify the application to add FPE code, so the question becomes whether that additional encryption/decryption code entails less effort and risk than an alternative modification, with the answer dependent on the existing environment and specifics of the FPE implementation under consideration. As many of you reading this document will be considering FPE for PCI compliance. Note that while many auditors accept FPE variants, at the time of this writing neither NIST nor the PCI council have officially certified FPE.

We recommend FPE for systems where the hardware and firmware cannot be altered to be used with traditional ciphers, or to restrict certain application level access in multi-node processing environments. Additionally, FPE ciphertext can transparently be used as tokens by systems which do not need the actual data, and decrypted (rather than looked up through a tokenization server) by applications which do need it. In essence you get tokenization where it’s suitable, but select applications can independently access the original data. Systems which do not need the sensitive data simply don’t get decryption capabilities or keys.

This flexibility comes at a price, as you still must deploy a secure key management system, and FPE does not fully remove sensitive data, which may affect the system’s compliance requirements. Additionally, risk is higher because multiple trusted systems have access to decryption keys or decrypted data. For organizations that need this type of flexibility, FPE accomplishes many of the same goals and is a reasonable substitute for tokenization. The key difference between the two is that FPE data can be decrypted by any system with the key, while tokenized data is only accessible through the master token server.

Key Management

Key management is a top concern for any encryption project. There are two basic options for key management; keys can reside inside the database, or outside. They support the same encryption algorithms, but have different implications for the database and applications. How keys are shared, archived, secured, and accessed are all different. Internal key management removes most of the complexity and management burden by allowing the database to provide key generation and encryption functions without changing application logic or data processing. External key management services offer superior management capabilities, with greater cost and complexity. Let's dig into some key management practices for databases and how they are used.

Internally Managed

For database encryption, we define "internal key management" as key services within the database, provided by the database vendor. All relational database management platforms provide encryption packages, including key management. Typical services include key creation, storage, retrieval, and security; most systems can handle both symmetric and public key encryption. Use of keys can be handled by proxy, as with the transparent encryption options, or through direct API calls to the database package. The keys are stored within the database, usually within a special table that resides in the administrative database or schema. Each vendor's approach to securing keys varies significantly from their competitors. Some vendors rely upon simple access controls tied to administrative accounts, some encrypt individuals' keys with a single master key, and others do not allow any access to keys at all, performing all key operations within a proxy.

If you are using transparent encryption options provided by the database vendor, all key operations are performed on the users' behalf. For example, when a query for data within an encrypted column is made, the database performs the typical authorization checks, and when authorized automatically decrypts the data for the user. Neither the users nor the application needs to be aware the data is encrypted, to make a specific request to decrypt, or to supply a decryption key. It is all handled automatically on their behalf, and often without their knowledge.

Transparent key management's greatest asset is just that: its transparency. The storage, management, security, sharing, and backup of the keys is handled by the database. With internally managed encryption keys, there is not much for the application to do, or even to care about, since all use and management of keys occurs inside the database. The service runs autonomously and is activated by simple alteration of database parameters, so the DBA need only worry about backing up the system and running the occasional disaster recovery drill so there are no unexpected 'gotchas'. The database vendors have worked hard to perfect these operations and made them fast, efficient, and reliable.

On the down side, keys are only as secure as your least trustworthy DBA. In any platform that stores the keys in a database table, they are accessible to database administrators. Despite vendor advertisements to the contrary, there are attacks that can sniff the key operations within the database, or even scan through database memory structures where

the keys reside during use. When you boil it down, transparent encryption with associated internal key management is a very simple and cost effective way to secure data at rest, but not effective for securing keys from database administrators or determined attackers who gain access to the host.

Externally Managed

With external key management, all key operations are performed outside the database in a dedicated key management server. The servers are dedicated software or hardware appliances (Hardware Security Modules, or HSMs), available through network or local application procedure calls. Both types provide a full range of functions including key creation, storage, and retrieval; they support both symmetric and public key cryptography. But they offer much more — including administrative support for access management, sharing keys across multiple key managers, and help with policies for key expiration/rotation. External key management servers provide additional advantages over internally managed keys because the platform is dedicated to this function and provides faster cryptographic operations, they produce stronger encryption (better keys), and they store keys more securely than is possible within the database. They can also perform the encryption and decryption operations outside the database, if desired.

External key management can be used in two ways: by calling the key management server directly, or by having the database make requests on behalf of users and applications. All relational databases now provide APIs for external cryptography. By calling the database through an established database communication port, application developers get external key management services within the same database connection. These API calls are performed in the same context and using the same programming interfaces as other database queries. This approach offers enforces separation of duties, keeps the encryption keys out of the hands of database administrators, and puts the keys under application developer control. Application developers can choose how granular they want to get with data encryption, and gain considerable flexibility in access control enforcement and key usage. That approach still leverages the database itself to perform the encryption, and can take advantage of database features to address many structural and systems management challenges.

The other option is for applications to call the key manager directly, and have it perform all cryptography on their behalf. In this case you are outside the realm of database encryption and have essentially moved to external application layer encryption. By performing all encryption outside the database, eschewing its built-in encryption capabilities, the database becomes a simple repository for encrypted data. The advantages are several: the database does not suffer the performance impact of encryption operations, the network latency of a database API call is not incurred, remote key storage enables separation of admin duties, and these facilities can be shared across many different systems. On the other hand, all cryptographic operations are now the responsibility of the application developer, all user and key management must be handled within the application logic, and the database design must be altered. This includes stored procedure code as well as table design.

External key managers handle key sharing, backup, and other services that would otherwise be handled automatically by the database itself. Most customers we speak with now opt for dedicated hardware to support key management operations. Performance is an advantage, as hardware acceleration can generally handle encryption tasks an order of magnitude faster than software routines on general-purpose (database) servers. Another advantage is high-quality entropy to seed cryptographic ciphers. Additionally, sharing of keys across servers is well secured, and the APIs can be accessed by multiple applications. Finally, HSMs are protected from physical tampering and electronic eavesdropping, making them the most secure key storage options discussed here. HSMs tend to be very expensive, compared to software (either DBMS features or software encryption applications).

That is not to say that there are no advantages to dedicated software key management. Simplicity is a major factor — as is support for multiple platforms, which provides some flexibility and conformity in hardware requisition and maintenance; but the biggest advantage of dedicated software key management is the ability to run anywhere. Running on generic hardware helps to rapidly recover from disasters, and it is likely to be difficult or even impossible to integrate HSMs into virtual environments. Performance, however, is inferior to dedicated hardware, and the keys are not as strongly secured as on dedicated devices with hardware security features.

Key Management & Threats

In this section we highlight some of the benefits, weaknesses, and practical realities that come into play with database encryption and key management. We have stressed throughout this paper that access controls are the first line of defense, and encryption generally is worthless if a user account is compromised. This is because for practical reasons any account must generally have automatic access to its keys, so successful compromise of an account or credentials normally includes the same decryption access as the legitimate user would have. Encryption keys provide much better resistance to brute force attacks than passwords. A principal motivation for using external key management and application level encryption is to avoid tying keys solely to accounts. Let's take a closer look at some of the vendor tools to see this in action, and how they protect from this kind of cascade failure.

With Microsoft SQL Server, implementing encryption at the application level, you can protect a key with a password or encrypt it with a different key. Assume we have a generic application account called 'SAPConnPool_User', used for creating a pool of connections to the database, with the password 'Password1'. Should an attacker guess the password 'Password1', they have not automatically been granted use of the decryption keys. Users must supply separate credentials to the 'DecryptByKey' command to decrypt the data. The application will pass keys in context of the query operations, on behalf of the user, meaning that the hacker cannot gain access to the data by guessing the password. A DBA can gain access to this key if it is stored inside the database, or even intercept API calls, which is why fully separated key management offers greater security. IBM DB2 Universal Database offers similar protection, as the user connection to the database uses one set of credentials, while access to encryption keys uses a second set of credentials. To gain access you need both sets.

Transparent Encryption does not normally offer this redundancy. Once a user is validated to the database, their session is supplied with an encryption key, and encryption operations are automatically mapped to their queries. The user may even automatically have access to the table that stores the key, without additional credentials for access. This varies greatly across platforms and DBMS versions, and is something to evaluate against your security requirements.

If you view the problem as protecting data when database accounts have been compromised, then any form of key management that ties keys to accounts is subject to failure. Encryption credentials in the application layer are far safer only if they leverage multiple authentication methods prior to allowing access to keys. Otherwise applications provide application users the same type of transparency that Transparent Encryption provides database users, so a breached application account will also bypass encryption credentials and access data stored in the database. Same problem, different layer.

There is one additional practical concern regarding key management and databases: When a database is restarted, it needs access to the keys before it can access the data. But this introduces a chicken-and-egg problem, as key access is secured by a password or another key. Some databases can obtain keys in an 'unattended' mode, meaning it happens automatically without human intervention. You must be careful that the database is not leaving keys or passwords exposed within the database or external text files to accomplish this task. The other option is attended mode, which means a person must be present to enter a password in order to decrypt keys in storage. In this scenario, you

must be careful that the people with the password are trusted and reliably available, and that the key manager does not yield keys to the administrator, only to the database. Either method can be made secure, but how secure depends upon the implementation.

Putting It All Together

Encrypting data within a database doesn't always present a clear-cut value proposition. Many of the features/functions of database encryption are also available through external tools, creating confusion as to why (or even whether) database encryption is needed. In many cases, past implementations have left DBAs and IT staff with fears of degraded performance and broken applications — creating legitimate wariness the moment some security manager mentions encryption. Finally, there is often a blanket assumption that database encryption disrupts business processes and mandates costly changes to applications, which isn't necessarily the case. To make good database encryption decisions, you'll first need to drill down into the details of what threats you want to address and how your data is used. Let's review our decision tree from The Selection Process above. Look at the two basic options for database encryption, and select the variation or "leaf of the tree" appropriate to your situation to determine what you need. Only then can you make an informed decision on which database encryption model best suits your situation, if any.

The following are several use cases for database encryption — two are specific customer situations, with the analysis processes they followed. Let's use the decision tree and walk through the decision-making process to show how it works.

Real Data, Virtual Database

Company A is a telephony provider with several million customers, and services user accounts through their web site. The company is considering virtualizing their server environment to reduce maintenance costs, adapt to fluctuations in peak usage, and provide more options for disaster recovery. The database is used directly by customers through a web application portal, as well as by customer support representatives through a customer care application. The database is periodically updated by the billing department through week-end batch jobs.

Company A is worried that if virtual images of the database are exported to other sites within the company or to service provider sites, those images could be copied and restored outside company control. The principal threat they are worried about is off-site data inspection or tampering with the images. As secondary goals they would like to keep key management simple, avoid introducing additional complexity to the disaster recovery process, and avoid an increased burden for day-to-day database management.

Actions Taken

In this scenario, a variant of transparent or external encryption would be appropriate. Since the threat is non-database users accessing data by examining backups or virtual images, transparent encryption protects against viewing or altering data through the OS, file system, or image recovery tools. Which variant to choose — external or internal — depends on how the customer wants to deploy the database. The deciding factors in this case are two-fold: Company A wants separation of duties between OS and database administrators, and in their virtualized environment the availability of disk

encryption cannot be guaranteed. Native database encryption is the best fit for them: it inherently protects data from non-credentialed users, and removes any reliance on the underlying OS or hardware. Further, additional computational overhead for encryption can be mitigated by allocation of more virtual resources.

Recommendations

While the data would not be retrievable simply by examining the media, a determined attacker in control of (copies of) the virtual machine images could launch many copies of the database, and has forever to guess DBA passwords to obtain the decryption keys stored within the database, but using current techniques this isn't a significant risk (assuming good passwords). Regardless, native transparent encryption is a cost-effective method to address the company's primary concerns without interfering with IT operations, with any weaknesses further reduced by external key management.

Near Miss Breach

Company B is a very large technology vendor, concerned about the loss of sensitive company information. During an investigation of missing test equipment from one of their QA labs, a scan of public auction sites revealed that not only had their stolen equipment been recently auctioned off, but several servers from the lab were actively listed for sale. With the help of law enforcement they discovered and arrested the responsible employee, but that was just the beginning of their concern. As the quality assurance teams habitually restored production data provided to them by DBAs and IT admins onto test servers to improve the realism of their test scenarios, a forensic investigation showed that most of their customer data was on the QA servers which had been up for auction. The data in this case was not leaked to the public, but the executive team was shocked to learn they had very narrowly avoided a major data breach, and decided to take proactive steps against sensitive data escaping the company.

Company B has a standing policy regarding the use of sensitive information, but understands the difficulty of enforcing this policy across the entire organization forever. The direct misuse of the data was not malicious — the QA staff were working to improve the quality of their simulations and indirectly benefiting end users by projecting demand accurately — but had the data leaked this fine distinction would be irrelevant. To help secure data at rest in the event of accidental or intentional disregard for data security policy, the management team has decided to encrypt sensitive content within these databases. The question becomes which option would be appropriate: user or transparent encryption.

This example was directly inspired by a real situation.

Actions Taken

The primary goal here is to protect data at rest, and secondary is to provide some protection from misuse by internal users. In this case, the company decided to use user-based encryption with key management internal to the database. Encrypted tables protect against data breach in case servers, backup tapes, or disks should leave the company; they also address the concern of internal groups importing and using data in non-secured databases.

Recommendations

At the time this analysis took place, the customer's databases were older versions that did not support separation of roles for database admin accounts. Further, the databases were installed under domain administration accounts — providing full access to both application developers and IT personnel; this access is integral to the data backup & archiving system. At the time tying the encryption keys to user and service accounts was considered an effective way to address the threats, and performance was superior to full database encryption because sensitive data was constrained to a few columns. This use case reflects a real customer, and how they chose to deal with the issue at the time. If the decision was made for a new deployment, the choice would be different. Transparent encryption, proper deployment, and the modification of access control settings would have been sufficient to remedy both problems and would be the

optimal choice today. But we must acknowledge that the optimal choice is not always available given IT environments and legacy platforms in use.

PCI Compliance Strategy

Company C is a Tier 2 merchant which needs to comply with PCI-DSS guidelines. They store customer billing information, credit card information, and some password recovery information in the customer database. Some transactional information with customer name and address information is also propagated to other databases, with foreign references from the customer table into the billing department database. The customer wants to comply with the PCI-DSS standard and would like to keep the data segmented from all users with the exception of a single administrative account and the lone service account which processes requests from the application server.

Actions Taken

The customer's decision was to break this into a two-phase effort. As they were behind the curve, the first goal was to attain PCI compliance, before migrating to a more secure and more sustainable solution. To get compliant quickly, they chose a file-based encryption product that externally secured database contents. This was implemented without alteration to the application and database logic. Company C decoupled access control from the native OS platform by leveraging an existing centralized service. This was a stop-gap measure, providing breathing room to move to a different architecture.

As a long-term solution, Company C is removing the use of credit card numbers from business processing applications, and moving to internal tokenization. Every credit card transaction will generate a token that is used in lieu of the credit card number. As this number is nothing more than an internal reference, it cannot be used as a credit card in the event it is stolen. All other internal operations that reference credit card numbers will be replaced with tokens provided by the internal software. As the tokens closely resemble the original credit card data, they will require few changes to supporting applications and none to database structure.

All credit card details will be moved into the single data repository, reducing the scope of the problem. Company C evaluated building their own solution, but decided to go with a commercial tokenization tool running on internal systems. Their tokenization solution runs on top of a dedicated database, and handles all the key management and encryption operations. The selection of a commercial solution reduced deployment time significantly, even though it required various application and database changes.

Recommendations

While this use case reflects a single company's strategic decision, many of the larger firms we have spoken to selected a similar tokenization strategy. But smaller firms are opting for total replacement of credit card data. Rather than keeping PAN and related information in a single database, it is more efficient for them to totally remove most liability by running transactions through a third party. If you can get rid of the data entirely or use tokens, these should be your first choice.

If you are looking to "get compliant fast" with PCI, we typically recommend some form of transparent encryption. Customer buying decisions are generally split between transparent encryption offered by the database vendor and transparent OS/file level encryption. Both protect media, offer seamless integration, and perform reasonably well in typical database environments. Both require additional expense, and we recommend you buttress either product with external key management, they meet most compliance requirements with minimal fuss. Which you choose will be determined by your needs, but typically transparent encryption is slightly easier to deploy and comes from a familiar vendor, whereas OS and file level encryption offers a slight performance gain and the flexibility to be used for non-

database security. OS and file level encryption can also be implemented on older databases that don't include transparent encryption as an option.

If you are implementing encryption as part of a new application, and not retrofitting an older environment, we recommend application level encryption. It offers the most flexibility and potentially the highest level of security. You can encrypt at the cell, row, or column level; you can choose any cryptographic libraries or key management system available; you get complete separation of duties between IT and database administrators; and you can disassociate access controls from key usage through secondary authentication and certificate verification.

Conclusion

Database encryption is one of the more complex data security issues. There are a myriad of options, each with different benefits and costs — some non-obvious. For example, one common belief is that encrypting a column provides better performance than encrypting an entire database at the file level, since you are encrypting less data. But in reality, column-level encryption often results in *worse* performance because it breaks many of the indexing, query, and relational tools used by the DBMS.

Over the past few years we've seen some important changes in the database security market, due mostly to improved offerings from the database vendors. We are seeing less focus on user-level column encryption as the security and database world shifts to two different strategies. For protecting against non-credentialed users, most organizations adopt transparent encryption, or use a third-party file-level tool to encrypt the database files. Choosing between those options usually comes down to price and database versions. This is also the most common option for quickly securing an existing database to meet compliance needs.

For financial systems, especially credit card processing, we see increasing use of application level encryption, sometimes combined with tokenization. In some cases it's built into the transaction software, while in others it is a larger — often multi-year — project.

Database encryption may be daunting at first, and might remain daunting if you're faced with a complex environment, but by understanding what you need to protect and what to protect it from, you can use the decision tree in this report to quickly select the best strategy.

Who We Are

About the Authors

Rich Mogull, Analyst/CEO

Rich has twenty years experience in information security, physical security, and risk management. He specializes in data security, application security, emerging security technologies, and security management. Prior to founding Securosis, Rich was a Research Vice President at Gartner on the security team, where he also served as research co-chair for the Gartner Security Summit. Prior to his seven years at Gartner, Rich worked as an independent consultant, web application developer, software development manager at the University of Colorado, and systems and network administrator. Rich is the Security Editor of *TidBITS*, a monthly columnist for *Dark Reading*, and a frequent contributor to publications ranging from Information Security Magazine to *Macworld*. He is a frequent industry speaker at events including the RSA Security Conference and DefCon, and has spoken on every continent except Antarctica (where he's happy to speak for free — assuming travel is covered).

Adrian Lane, Analyst/CTO

Adrian Lane is a Senior Security Strategist with 22 years of industry experience, bringing over a decade of C-level executive expertise to the Securosis team. Mr. Lane specializes in database architecture and data security. With extensive experience as a member of the vendor community (including positions at Ingres and Oracle), in addition to time as an IT customer in the CIO role, Adrian brings a business-oriented perspective to security implementations. Prior to joining Securosis, Adrian was CTO at database security firm IPLocks, where he was responsible for product and technology vision, market strategy, PR, and security evangelism. Mr. Lane also served as Vice President of Engineering at Touchpoint, for three years as CIO of the brokerage CPMi, and for two years as CTO of the security and digital rights management firm Transactor/Brodia. Mr. Lane is a Computer Science graduate of the University of California at Berkeley with post-graduate work in operating systems at Stanford University.

About Securosis

Securosis, L.L.C. is an independent research and analysis firm dedicated to thought leadership, objectivity, and transparency. Our analysts have all held executive level positions and are dedicated to providing high-value, pragmatic advisory services.

We provide services in four main areas:

- Publishing and speaking: Including independent and objective white papers, webcasts, and in-person presentations.
- Strategic consulting for end users: Including product selection assistance, technology and architecture strategy, education, security management evaluation, and risk assessment.
- Strategic consulting for vendors: Including market and product analysis and strategy, technology guidance, product evaluations, and merger and acquisition assessment.
- Investor consulting: Technical due diligence including product and market evaluations, available in conjunction with deep product assessments with our research partners.

Our clients range from stealth startups to some of the best known technology vendors and end users. Clients include large financial institutions, institutional investors, mid-sized enterprises, and major security vendors.

Securosis has partnered with security testing labs to provide unique product evaluations that combine in-depth technical analysis with high-level product, architecture, and market analysis.