Chapter 2

Securing Databases with Cryptography

This chapter discusses how cryptography can address the concerns raised in the previous chapter. After explaining what cryptography is and providing a general idea of how it works, we dig into the various types of cryptographic algorithms and see where the strengths and weaknesses of each lie.

Finally, we look at where database cryptography sits in an organization's security portfolio. With respect to threats against confidentiality and integrity, we examine how cryptography can help with security. We also look at the common pitfalls and difficulties encountered in implementing a cryptographic system. Not only does a poorly implemented system not provide the needed protection, it can actually weaken overall security. We spend time looking at what kinds of risks a poor cryptographic system introduces.

2.1 A Brief Database Refresher

For the most part, this book assumes knowledge of databases, but we'll quickly go over the fundamentals in case you've been away from the topic for some time. A relational database stores information in tables consisting of rows and columns. A field or cell is the intersection of a row and column.

Tables are related to each other through primary and foreign keys (these keys are quite different from cryptographic keys, which are discussed later). A primary key is a subset of the information in a row that uniquely identifies that row from all the other rows in the table. A foreign key links a row in one table to a row in another table by referencing the latter table's primary key.

Indexes allow for quick searching through a table. By specifying an index on a column, the database creates a special data structure that allows it to rapidly find any information stored in that column. Primary key columns are typically indexed.

A standard language, structured query language (SQL), is used to manage data. Database objects, such as tables and indexes, are created, modified, and destroyed using a subset of SQL known as data definition language (DDL). Information is entered, viewed, altered, and deleted from a database using another subset of SQL called data manipulation language (DML).

The most common interaction with a database is the `select` statement, which is an element of DML. The `select` statement allows an operator to dig though one or more database tables and display just the data that meets specific criteria. A basic `select` statement contains three clauses. The *select* clause specifies which columns should be displayed. The *from* clause specifies which tables should be included in the search. The *where* clause details the criteria a row must meet to be selected.

The where clause frequently contains *join* statements, which tell the database how to include multiple tables in the query. Typically, a join follows the link established by a foreign key.

Other frequently used statements include `insert`, for inserting new data into a table; `update`, for modifying existing data in a table; and `delete` for removing rows. All of these statements also include from and where clauses.

Programs typically interact with databases by building and passing these statements to the database. For instance, when a customer wishes to see the items she added to her shopping cart last week, the application passes a `select` statement to the database to select all of the items in that customer's cart. Then, when the customer adds an item, the application might pass an `insert` to the database.

Stored procedures offer another avenue for an application to interact with a database. A stored procedure is a program that is loaded into the database itself. Then, instead of the application building an `insert` statement to add a new item to the customer's cart, the application would call the `add_item_to_cart` stored procedure and pass the item and quantity as arguments.

Databases are much more complex and feature-rich than what we've described here, but this overview should provide enough context to help you make sense of the database terminology used in this book. The code examples at the end of the book contain many examples of SQL statements. See Chapter 21, "The System at Work," for example.

2.2 What Is Cryptography?

Cryptography is the art of “extreme information security.” It is extreme in the sense that once treated with a cryptographic algorithm, a message (or a database field) is expected to remain secure even if the adversary has full access to the treated message. The adversary may even know which algorithm was used. If the cryptography is good, the message will remain secure.

This is in contrast to most information security techniques, which are designed to keep adversaries away from the information. Most security mechanisms prevent access and often have complicated procedures to allow access to only authorized users. Cryptography assumes that the adversary has full access to the message and still provides unbroken security. That is extreme security.

A more popular conception of cryptography characterizes it as the science of “scrambling” data. Cryptographers invent algorithms that take input data, called *plaintext,* and produce scrambled output. Scrambling, used in this sense, is much more than just moving letters around or exchanging some letters for others. After a proper cryptographic scrambling, the output is typically indistinguishable from a random string of data. For instance, a cryptographic function might turn “Hello, whirled!” into `0x397B3AF517B6892C`.

While simply turning a message into a random sequence of bits may not seem useful, you’ll soon see that cryptographic hashes, as such functions are known, are very important to modern computer security. Cryptography, though, offers much more.

Many cryptographic algorithms, but not all, are easily reversible if you know a particular secret. Armed with that secret, a recipient could turn `0x397B3AF517B6892C` back into “Hello, whirled!” Anyone who did not know the secret would not be able to recover the original data. Such reversible algorithms are known as *ciphers*, and the scrambled output of a cipher is *ciphertext*. The secret used to unscramble ciphertext is called a *key*. Generally, the key is used for both scrambling, called *encryption*, and unscrambling, called *decryption*.

A fundamental principle in cryptography, Kerckhoffs’ Principle, states that the security of a cipher should depend only on keeping the key secret. Even if everything else about the cipher is known, so long as the key remains secret, the plaintext should not be recoverable from the ciphertext.

The opposite of Kerckhoffs’ Principle is security through obscurity. Any cryptographic system where the cipher is kept secret depends on security through obscurity.[1] Given the difficulty that even professional cryptographers have in designing robust and efficient encryption systems, the likelihood of a secret cipher providing better security than any of the well-known and tested ciphers is vanishingly small. Plus, modern decompilers, disassemblers, debuggers, and other reverse-engineering tools ensure that any secret cipher likely won't remain secret for long.

Cryptographic algorithms can be broadly grouped into three categories: symmetric cryptography, asymmetric (or public-key) cryptography, and cryptographic hashing. Each of these types has a part to play in most cryptographic systems, and we next consider each of them in turn.

2.2.1 Symmetric Cryptography

Symmetric key cryptography is so named because the cipher uses the same key for both encryption and decryption. Two famous ciphers, Data Encryption Standard (DES) and Advanced Encryption Standard (AES), both use symmetric keys. Because symmetric key ciphers are generally much faster than public-key ciphers, they are suitable for encrypting small and large data items.

Modern symmetric ciphers come in two flavors. *Block ciphers* encrypt a chunk of several bits all at once, while *stream ciphers* generally encrypt one bit at a time as the data stream flows past. When a block cipher must encrypt data longer than the block size, the data is first broken into blocks of the appropriate size, and then the encryption algorithm is applied to each. Several *modes* exist that specify how each block is handled. The modes enable an algorithm to be used securely in a variety of situations. By selecting an appropriate mode, for instance, a block cipher can even be used as stream cipher.

The chief advantage of a stream cipher for database cryptography is that the need for padding is avoided. Given that block ciphers operate on a fixed block size, any blocks of data smaller than that size must be padded. Stream ciphers avoid this, and when the data stream ends, the encryption ends. We'll return to block and stream ciphers in the algorithm discussion in Chapter 4 "Cryptographic Engines and Algorithms."

The primary drawback of symmetric key ciphers is key management. Because the same key is used for both encryption and decryption, the key must be distributed to every entity that needs to work with the data. Should an adversary

1. A military cipher would seem an exception, except that most likely the cipher is designed in accordance with Kerckhoffs' Principle in the not-too-unlikely case that an enemy discovers the cipher.

obtain the key, not only is the confidentiality of the data compromised, but integrity is also threatened given that the key can be used to encrypt as well as decrypt.

The risks posed by losing control of the key make distributing and storing the key difficult. How can the key be moved securely to all the entities that need to decrypt the data? Encrypting the key for transmission would make sense, but what key would be used to encrypt the key, and how would you get the key-encrypting key to the destination?

Once the key is at the decryption location, how should it be secured so that an attacker can't steal it? Again, encryption offers a tempting solution, but then you face the problem of securing the key used to encrypt the original key.

We'll look at these problems in more detail in Chapter 5 "Keys: Vaults, Manifests, and Managers." In terms of the key distribution problem, cryptographers have devised an elegant solution using public-key cryptography, which we examine next.

2.2.2 Public-Key Cryptography

Public-key cryptography, also known as asymmetric cryptography, is a relatively recent invention. As you might guess from the name, the decryption key is different from the encryption key. Together, the two keys are called a key pair and consist of a public key, which can be distributed to the public, and a private key, which must remain a secret. Typically the public key is the encryption key and the private key is the decryption key, but this is not always the case. Well-known asymmetric algorithms include RSA, ElGamal, and Diffie-Hellman. Elliptic curve cryptography provides a different mathematical basis for implementing existing public-key algorithms.

Public-key ciphers are much slower than symmetric-key ciphers and so are typically used to encrypt smaller data items. One common use is to securely distribute a symmetric key. A sender first encrypts a message with a symmetric key and then encrypts that symmetric key with the intended receiver's public key. He then sends both to the receiver. The receiver uses her private key to decrypt the symmetric key and then uses the recovered symmetric key to decrypt the message. In this manner the speed of the symmetric cipher is still a benefit, and the problem of distributing the symmetric key is removed. Such systems are known as hybrid cryptosystems.

Another important use for public-key cryptography is to create digital signatures. Digital signatures are used much like real signatures to verify who sent a message. The private key is used to sign the message, and the public key is used to verify the signature.

A common, easily understood digital signature scheme is as follows. To sign a message, the sender encrypts the message with the private key. Anyone with the corresponding public key can decrypt the message and know that it could only have been encrypted with the private key, which presumably only the sender possesses. Note that this does not protect the confidentiality of the message, considering anyone could have the sender's public key. The goal of a digital signature is simply to verify the sender.

Because the public key can be distributed to anyone, we don't have the same problem as we do with symmetric cryptography. However, we do have a problem of unambiguously matching the public key with the right person. How do we know that a particular public key truly belongs to the person or entity we think it does? This is the problem that public key infrastructure (PKI) has tried to solve. Unfortunately, PKI hasn't lived up to its promise, and the jury is still out on what the long-term accepted solution will be.

Public-key cryptography is mentioned here to help readers new to cryptography understand how it is different from symmetric algorithms. We do not use public-key cryptography in this book, and we do not cover particular algorithms or implementation details. As is discussed in section 2.3, "Applying Cryptography," public-key schemes aren't necessary for solving the problems in which we're interested.

2.2.3 Cryptographic Hashing

The last type of cryptographic algorithm we'll look at is cryptographic hashing. A cryptographic hash, also known as a *message digest,* is like the fingerprint of some data. A cryptographic hash algorithm reduces even very large data to a small unique value. The interesting thing that separates cryptographic hashes from other hashes is that it is virtually impossible to either compute the original data from the hash value or to find other data that hashes to the same value.

A common role played by hashing in modern cryptosystems is improving the efficiency of digital signatures. Because public-key ciphers are much slower than symmetric ciphers, signing large blocks of data is very time-consuming. Instead, most digital signature protocols specify that the digital signature is instead applied to a hash of the data. Given that computing a hash is generally fast and the resulting value is typically much smaller than the data, the signing time is drastically reduced.

Other common uses of cryptographic hashes include protecting passwords, time-stamping data to securely track creation and modification dates and times, and assuring data integrity. The well-known Secure Hash Algorithm family includes SHA-224, SHA-256, SHA-384, and SHA-512. The older SHA-1 and MD5 algorithms are currently in wider use, but flaws in both have been identified, and both should be retired in favor of a more secure hash.

2.3 Applying Cryptography

Now that you've freshened your recollection of database terminology and surveyed the basics of modern cryptography, we examine how cryptography can help secure your databases against the classes of threats covered in Chapter 1.

As we discuss the types of solutions offered by cryptography, we'll also consider the threats that cryptography is expected to mitigate. This threat analysis, as discussed previously, is an essential component of any cryptographic project, and the answers significantly shape the cryptographic solution. Unfortunately, in practice, a requirement to encrypt data is rarely supported with a description of the relevant threats. Encrypting to protect confidentiality from external attackers launching SQL injection attacks is different from protecting against internal developers with read-only access to the production database. The precise nature of the threat determines the protection.

2.3.1 Protecting Confidentiality

A breach of confidentiality occurs when sensitive data is accessed by an unauthorized individual. Encrypting that sensitive data, then, seems to make excellent sense: if the data is encrypted, you've secured it against unauthorized access. Unfortunately, the solution is not this simple. Cryptography only changes the security problem; it doesn't remove it.

The initial problem was to protect the confidentiality of the business data. Encrypting that data changes the problem to one of protecting the confidentiality of the key used for the encryption. The key must be protected with very strong access controls, and those controls must cover both direct and indirect access.

Direct access is access to the key itself. An attacker with direct access may copy the key and use it without fear of detection. Indirect access is access to an application or service that has access to the key. With indirect access, an attacker can feed encrypted data to the application or service and view the decrypted information. An attacker exploiting indirect access faces additional risk, because the application or service, due to its sensitivity, is generally well monitored. From an attacker's point of view, the advantage that might make the indirect access worth the additional risk is that the application or service will continue to provide decryption even after the key is changed. An attacker who has a copy of the key will find the key useless as soon as the data is encrypted with a different key.

The problem of securing access to the key lies behind much of the complexity of key management systems. Cryptography is often said to transform the problem of protecting many secrets into the problem of protecting one secret. Protecting this one secret, despite the complexity, is generally easier than protecting the many secrets. Because of this, encryption is a strong and preferred method of protecting confidentiality in a database.

Consider the confidentiality threats identified in the previous chapter. The potential attackers included individuals with a copy of the database, privileged administrators, IT troubleshooters, development staff using nonproduction environments, individuals with access to backup media (often stored off-site), and attackers exploiting application weaknesses. In each of these cases, encryption protects the data so long as access to the keys is tightly controlled.

While tightly controlling direct access to keys is a relatively solvable problem (this book recommends dedicated key storage hardware but offers suggestions if such protection is unavailable), controlling indirect access is more difficult. Information is stored because the business is likely to need it at a later date, and to use it at that later date, encrypted information will need to be decrypted. The application that provides the decryption service is a weak link in the security chain. Rather than attack the encryption or the controls protecting direct access to the key, a smart attacker targets the decrypting application. We'll consider this issue in more detail in section 2.5.1, "Indirect Access of Keys."

Protecting against attackers with access to a copy of the database, whether stolen from a production machine or a backup tape, requires that the key not be stored within or, ideally, on the same machine as the database. In the case of attacks exploiting backup media, backups of keys must be stored separately from the backups of encrypted data. Access to those backups must also be restricted to different individuals. How deep that separation goes depends on the threat model. If the relevant threat is the off-site backup staff, two different staffs (perhaps two different backup companies) are all that is necessary. If the relevant threats extend all the way to internal system administrators, separate administrators should manage the backups of each system.

Protecting against administrators with full access to database or application servers is primarily a matter of strong indirect access controls to the keys. However, even with just moderate access controls protecting the keys, encryption prevents casual attacks from administrators. In particular, encryption significantly increases the amount of effort required for an administrator to compromise confidentiality and keep the risk of detection low. Such protection is often described as "keeping the honest, honest." If the threat model rates the risk of administrator compromise sufficiently low, keys may need only a moderate level of protection from indirect access.

Most threat models should identify the presence of sensitive production data in nonproduction environments as a significant threat. Because encryption does such a fine job of preserving confidentiality, encrypted production data can't be decrypted in a nonproduction environment.[2] While this is good from a security perspective, the failure in decryption typically results in a malfunctioning environment.

The best solution is to replace all the encrypted data with mock data. Ideally, the mock data reflects the entire data model specified by the application's design, including common, general-use scenarios as well as edge cases. Depending on resource availability, the mock data might be encrypted after it is written to the database. In some cases, it might be possible to encrypt the mock data first and then update the table with the encrypted mock data wherever it is needed. This latter strategy avoids the row-by-row encryption of the mock data.

2.3.2 Assuring Integrity

Cryptography can help detect and prevent integrity attacks, which are unauthorized modifications of data. In some cases, both integrity and confidentiality protection are desired, while in other cases just integrity protection may be needed. When integrity protection alone is called for, the data itself remains unencrypted, and some other operation protects integrity.

The naive solution for both confidentiality and integrity is to simply encrypt the information with a symmetric cipher. Later, if it doesn't decrypt properly, someone has tampered with the information.

Unfortunately, the naive solution is not very robust. A clever attacker will attack integrity in a less obvious fashion. For instance, an attacker might move encrypted fields around so that the rows containing my information now have someone else's encrypted credit card number. Or the attacker might swap blocks within the ciphertext or between ciphertexts. In such an attack, much of the field could decrypt to the correct value, but selected portions of it would decrypt to something else. This attack might result in some garbled data, but the rest of the field would look fine.

2. This assumes that the keys for decrypting the data are not available in nonproduction environments; if they are, the key management procedures are seriously flawed. This is discussed further in Part II, "A Cryptographic Infrastructure."

A better solution, and one that works even if confidentiality protection is not needed, is to use a message authentication code (MAC). A MAC is generated from the plaintext and a unique ID for that row (the ID thwarts attacks that move entire fields around). To confirm the integrity of the data, we check to make sure that the MAC still corresponds to the data.

While this will detect a past integrity attack, a MAC can also prevent integrity attacks. When data and its MAC are inserted into a table, the database can first check to ensure that the MAC is the correct MAC for that data. If it is the wrong MAC (or the MAC is not included), the database can reject the change.

Every database threat model should consider integrity threats, but as described in the previous chapter, cryptographic integrity protection is typically not a good fit for databases. The threat model will help make this clear. Integrity threats against the database may be carried out by attackers directly targeting the database or by attackers targeting the application providing access to the database, which also stamps changes with the MAC. In general, attacks against the application are more likely to be successful than attacks directly against the database, so, in this context, the risk posed by the application is greater than the risk posed by the database itself. To be effective, security should be applied to the higher-risk items. Because the protection offered by a MAC further increases the difficulty of directly attacking the database successfully (which is already a lower risk), those resources should be applied to securing the application instead, thus reducing the overall risk. Refer to section 1.1.3, "Integrity Attacks," for a more detailed discussion.

2.4 Cryptographic Risks

Cryptography should not be undertaken lightly. While it can significantly help secure information in a database, cryptography carries risk as well.

Perhaps the most obvious risk is the danger of lost keys. Should a key become lost, either corrupted or deleted or even accidentally thrown away, any data encrypted with that key is also lost. There is no "undelete" or "data recovery" program that can undo the encryption. All it takes is the loss of just 128 bits (the recommended size of a key in this book), and megabytes of data become meaningless. This threat is one of the reasons that key management is such an important topic in cryptography.

As mentioned earlier, weaknesses in key management tools and procedures can put overall security at risk. If an attacker can access the key, directly or indirectly, or insert a known key into the system, the cryptography is broken.[3] If the key generation routines aren't based on sufficiently random numbers, the attacker may be able to guess the key.

Implementation bugs also introduce risks. If other data used in the encryption process, such as initialization vectors, which are covered later, does not possess the appropriate properties, attackers will likely be able to discern patterns in the encrypted data and possibly deduce the real data. If the data is written to logs or even not wiped from memory, it is vulnerable to attackers. Even if the key management is perfect and the implementation bug-free, indirect access to the keys is still a significant issue.

Because poor encryption looks so similar to good encryption, it generates misplaced confidence, which can amplify the risks posed by the data. An encrypted system may not have as many other controls placed around it, so any vulnerabilities are even more exposed. In this way, a bad cryptographic system can decrease the data's security.

It is vitally important that the cryptographic infrastructure be designed and implemented correctly. Later chapters go into detail on the design of a cryptographic infrastructure.

2.5 Cryptographic Attacks

Cryptographers classify attacks against cryptosystems into several categories. These attacks attempt to either retrieve the key or expose the plaintext. The algorithms discussed in this book are strong and resist all the attacks discussed here. However, the demands of a practical cryptosystem can easily introduce vulnerabilities even though the algorithm itself is strong. Much of the design presented in this book is aimed at mitigating these weaknesses.

A *known-ciphertext* attack is what most people think of as a cryptographic attack. The attacker has access only to the ciphertexts produced with a given key. These attacks can target either the key or the plaintext. Generally, we'll assume that the attacker has all the ciphertexts.

In the case of a database, this is tantamount to the attacker's having access to the database. Perhaps the attacker has found a weakness in the operating system that allows the database file itself to be downloaded, or perhaps a SQL injection

3. Adding to the risk is the fact that such situations can be very difficult to detect.

attack is exposing the encrypted data. A properly placed insider often has easy access to all the data.

When the attacker has access to both the plaintext and the ciphertext, the attacker can mount a *known-plaintext* attack. People new to cryptography often dismiss known-plaintext attacks as a sort of "cheating." After all, if the attacker already has all the plaintexts, all the secrets have been exposed. We generally assume, though, that only some of the plaintext-ciphertext pairs are known. Perhaps all the past plaintexts prior to a certain date were compromised. The goal of a known-plaintext can be to recover the key or to uncover plaintext.

In a database context, it is often not too hard to find known plaintexts. The system might temporarily cache the plaintext prior to encryption, or the system might store the data unencrypted elsewhere in the system. This last case is far more common than you might think. For instance, say customer data is stored encrypted, but the data is decrypted in order to e-mail the invoice. The invoice might very well be stored in the database as well. If the invoice isn't also encrypted, the attacker has a source of plaintexts to match with ciphertexts.

An even more subtle example is when data taken together must be encrypted but when the data is separate, it can be unencrypted. For instance, a customer's name and credit card number might be encrypted when they are together in the order table. But another table, in the call tracking system, perhaps, might have the customer's name unencrypted. If these two tables can be linked in a series of joins, the attacker has access to the plaintext. Database normalization can help security in this case, but in practice many databases are not highly normalized, so leaks like this are common.

As its name implies, a *chosen-plaintext* attacker can construct plaintext for the system to encrypt. This is a much more powerful version of a known-plaintext attack. An even more powerful variation is when the attacker can experiment by constructing new plaintexts based on the results of previously constructed plaintexts.

This attack is generally quite easy to mount against a database. In the case of an online ordering system, the attacker simply places additional orders with whatever data he would like to see encrypted. If he would like to see the ciphertext for "Kenan," placing a false order with that information would be suffucient. Unless the cryptosystem is designed carefully, the attacker would then be able to identify all the rows in the table with an order for "Kenan" (and encrypted with a particular key) by searching for the ciphertext produced by the chosen-plaintext attack.

2.5.1 Indirect Access of Keys

The general strategy to protect against direct access of keys is to design the cryptosystem to ensure that the keys are never available outside the code that uses them. Ideally, the keys are locked in dedicated tamper-proof hardware that also contains the cryptographic code. Indirect access, though, as discussed earlier, is a much thornier problem since programs must be able to decrypt the data in order to process it.[4] If automatically launched programs can decrypt the data, a sufficiently motivated and skilled attacker will eventually be able to do the same.

In practice, an indirect access of keys is typically made through a function that passes data to the cryptosystem for decryption. This decryption function is often the weakest link in the security chain protecting encrypted data. If compromised, it will enable an attacker to decrypt arbitrary ciphertexts.[5]

Ideally, the cryptosystem is guarded by strong access controls that require authentication and authorization for each decryption call. To make this effective, though, the authorization check needs to occur as close to the actual decryption as possible. If a dedicated cryptographic device (discussed in Chapter 4, "Cryptographic Engines and Algorithms") is in use, the device should make the check itself. Unfortunately, that capability is very rare. The goal of these measures is to prevent the attacker from following the chain of function calls until a decryption call is found *after* the authorization check. If such a call is found, the attacker uses it for the decryption compromise attack.

Automated processes throw a wrinkle into this strategy. Automation, such as a batch credit card payment process, often needs access to decrypted data, and while it is certainly possible to require the automation to provide credentials prior to decrypting data, those credentials must also be stored and protected. The following discussion of data obfuscation covers this situation in more detail, but it is best to assume that if the attacker is sophisticated enough to break the application sufficiently to access the decryption function, he also will be capable of retrieving any credentials used by the automation.

Our approach is one of containment and observation. First, we ensure that the decryption function decrypts only the fewest necessary columns. This contains the damage in the case of a decryption compromise; the attacker won't be able to

4. Do not store unneeded data, especially unneeded sensitive information such as customer data. Storing and protecting unneeded data increases your risk and costs and provides no business value.

5. A decryption compromise exploiting indirect key access is related to the chosen-ciphertext *class of attacks. The primary difference is that the general goal of a chosen-ciphertext attack is to recover the key, whereas a decryption compromise generally ignores attempts to recover the key (our algorithms are not susceptible to chosen-ciphertext attacks) and is content with decrypting all encrypted information in the system.*

decrypt all protected data in the system. Our next layer of defense, observation, is the critical control.

Extensive logging of decryption requests will expose sudden spikes caused by an attacker. Correlation of the logs with server processing and customer request logs helps reveal skimming.[6] Honeycombing is also a valuable technique against an attempted decryption compromise.

Honeycombs are to applications and databases what honeypots are to networks. A honeycombed application has several APIs that are named and look like they should do what the attacker wants, but in reality they simply trigger alerts. A honeycombed database contains tables and views that look promising to potential attackers, but any `select` against them triggers an alert. In some cases, honeycombs can take the form of special rows in a table that look legitimate, but the data is fake and any query against them results in an alert. Any alert fired by a honeycomb is by definition suspicious since the honeycomb serves no actual business purpose and there is no reason in the normal course of business that the honeycomb would be accessed.

2.6 Obfuscation

Cryptography can be applied so as to provide strong security or weak security. Generally, since cryptography is expensive and introduces risk, you want only as much security as is necessary. This book uses the term *obfuscation* to describe situations where encryption is used to provide a minimal amount of security. Obfuscation is used simply to make reading data more difficult and thereby prevent casual attacks.

Obfuscation should be used rarely. Strong encryption, not obfuscation, is necessary for nearly every threat requiring a cryptographic solution. Obfuscation is generally appropriate only as a solution of last resort and relies on other controls for adequate protection. For instance, consider the situation described in the previous section, where a program must, in order to decrypt data, log into a dedicated decryption machine over the internal network. The dedicated machine requires a password, so the program must have access to that password. The problem of how to protect that password is an issue.

Encrypting the password makes sense, but how would it be decrypted? The strong cryptographic solution would be to use the dedicated decryption machine,

6. Skimming is where the attacker grabs only a few items at a time as opposed to going for bulk. It is a tactic more appropriate for an insider, who typically has a longer attack window.

but that won't work because the decrypted password is necessary to access the machine. Another option would be to encrypt the password with a key stored elsewhere and not use the dedicated machine. Then the problem becomes a matter of protecting that key while also making it available to the program when it needs to use the password.

Complicated schemes where the credentials are entered manually by a security administrator when the program first starts and are then stored only in primary memory, perhaps split into several pieces, further shift the problem to one of protecting the key when it is in the administrator's possession. Such measures also increase the program's fragility and introduce dependencies for restarting the program. Additionally, the security of such techniques is, ultimately, questionable. A skilled and patient attacker with the right tools and access can simply wait until the program decrypts the password, at which point the attacker picks it out of memory. While this isn't trivial, it certainly isn't impossible. The threat model helps you determine if such a complex scheme is appropriate in a given situation.

Alternatively, the password could simply be obfuscated. An obfuscated password might be encrypted and stored in one file while the key used for the encryption is stored in another file. Anybody simply browsing through files won't accidentally compromise the password, but, obviously, if that individual were dedicated to retrieving the password, it would not be difficult to get both the encrypted password and the key. Various schemes could be employed to increase the difficulty, but by calling the protection obfuscation, we recognize that the security is rather easily broken, so we are reminded to maintain other controls appropriately. In this case, we want to ensure that only administrators and the necessary programs have access to both files.

2.7 Transparent Encryption

Several products on the market advertise transparent encryption. The name itself should raise a few questions: what kind of security does *transparent* encryption provide? The assumption is, of course, that the cryptography is transparent to the legitimate, authorized user but not to attackers. The security in such a system depends not only on the cryptosystem and key storage, but also on how well legitimate users can be distinguished from attackers.

Attackers are notoriously good at looking like legitimate users. Internal attackers typically *are* legitimate users. Many of the attacks discussed in the previous chapter are not stopped by transparent encryption, including SQL injection and "sanctioned" backdoors. The threat model must be considered carefully to see if transparent encryption offers the necessary protection.

For instance, triggers and views (features of some databases) can offer automatic encryption and decryption for authorized users. The trigger in this case is a program embedded in the database that runs every time a row is inserted and encrypts the necessary data. The view is a "virtual" table that decrypts the data before returning it to the requester. A `select` against the view (by an authorized user) returns decrypted data.

Which threats does this system protect against? Assuming that the keys are secured on another machine,[7] the risk posed by thievery of the database or backups is mitigated. Of course, if an attacker can steal the database file, it is not too much of a stretch to assume that he can attack the transparent encryption system itself or the database's authorization mechanism.

Also, threats from legitimate users with direct access to the database via an account that allows them to `select` against the encrypted table but not against the auto-decrypting view are reduced. However, such users are rare. Development staff engaged in troubleshooting is perhaps the most likely scenario, but, depending on the problem, they may need access to the auto-decrypting view. Most other users will access the data through an application that typically uses its own account, so they will not directly access the database. This is a rather minimal gain.

Most of our other threats still pose a risk for the database. Administrators can easily don the "authorized" user role and have the cryptosystem happily decrypt the data. Application crackers have the option of attacking the application or the database. If either succeeds, the attacker will be well on the way to accessing the encrypted data.[8] Application subversion attacks, such as SQL injection, will also likely remain viable since such attacks generally leverage the application's access privileges, and the application will likely access the database through an account that is authorized to view the decrypted data.

In a nutshell, the trigger and view solution described here reduces the data's online security to the security level of the passwords to the accounts that are authorized to select against the auto-decrypting view. Should any of those accounts be compromised, the cryptography is circumvented. In terms of online threats, if this level of security were acceptable, cryptography in the database wouldn't be necessary in the first place.

7. If the keys are actually stored in the same database as the encrypted data or even on the same machine as the database, the solution provides very little protection against all but the most simple threats.

8. One of the design goals of a secure system is to minimize the number of attack points, and this book attempts to remove the database as an attack point.

If the local threat model consists of only offline attacks, transparent encryption, such as the scheme just discussed, might be sufficient.[9] The threat model considered in this book, though, is more extensive. To mitigate online threats, we can, at the very least, require that an attacker gain access to both the database for the encrypted data *and* a cryptography service to decrypt it. The cryptography service can then be protected with much better security controls. We can better limit which machines may communicate with it, monitor it more closely, significantly strengthen its access credentials, increase the frequency with which the credentials are changed, and better protect the handling of those credentials.

When considering a transparent encryption solution, an organization's security analysts must carefully compare the threat model the solution targets with the organization's threat model. In addition, key storage must be considered. Ideally, the solution will support the use of a dedicated cryptographic hardware module for protecting the key and performing cryptographic operations. At the very least, the keys should be kept and used on a different machine.

A final word of caution. As you'll see later, using a good algorithm is not enough. How it is used is of equal importance. This *how* is captured in the *mode*, and selecting the right mode depends on many conditions. While no single mode is correct in every situation, some modes are very rarely the right choice. Always ask which mode is used, and be very suspicious if a vendor indicates that no mode was used or if electronic code book (ECB) mode was used. In either case, the system is likely to be vulnerable to chosen-plaintext attacks, as described earlier in this chapter. This is covered in more detail in Chapter 4.

2.8 Summary

This chapter looked at database cryptography as a hardening solution to what is often referred to as the "soft, chewy center" of most organizations. This hardening is the last line of defense between data and attackers.

The chapter opened with introductory coverage of databases and cryptography. The cryptographic overage included symmetric cryptography, asymmetric (or public-key) cryptography, and cryptographic hashing. While the material certainly won't turn you into a database expert or cryptographer, it should provide enough background for you to follow the rest of the book.

9. *If the threat model is concerned only about access of backup media, a dedicated backup-only encryption solution might be most appropriate. Such a scheme is generally much easier to implement than cryptography* in *the database.*

With the introductory material out of the way, the discussion turned to examining, at a high level, how cryptography can be applied to protect the confidentiality and integrity threats identified in the previous chapter. Confirming the principle that security is always a balance of trade-offs, the risks of attacks against a database cryptosystem were considered. In particular, we explored the idea that encryption turns the problem of protecting the confidentiality of a large quantity of business data into the problem of protecting a small set of keys.

This chapter also discussed obfuscation, the purposeful use of poor key protection to obtain a minimal amount of data security, and transparent encryption, the problematic technique of automatically decrypting information for any "legitimate" user.

This chapter hopefully provided a taste of what cryptography can and cannot do. With the promise and limits of cryptography covered, we now move on to exploring the details of a functioning cryptosystem.