Data Encryption 101: A Pragmatic Approach to PCI Compliance

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 Prime Factors Inc.

About Prime Factors:

Since 1981, Prime Factors has provided flexible, cost-effective encryption software to more than 1,000 customers world-wide. EncryptRIGHT, their newest offering, provides a better way to achieve PCI encryption compliance by simplifying key management and reducing the cost of deployment. EncryptRIGHT was designed from the ground up to meet security threats, accelerate deployments, and satisfy auditors that sensitive data is safeguarded. EncryptRIGHT runs on all major operating systems, provides broad application development support and can be implemented by programmers and non-programmers alike.

For more information, please visit www.primefactors.com/products/EncryptRight/index.html.

Copyright

Contributors

The following individuals contributed significantly to this report through comments on the Securosis blog and follow-on review and conversations:

Dan

Jay Jacobs

Kevin Kenan

Table of Contents

Introduction

The Payment Card Industry Data Security Standard (PCI-DSS) is a set of requirements on how to secure credit card numbers, but the specification serves more as general technical guidance than an operational checklist. Requirements are fairly general in nature, telling you what you *need to do* but not *how to do it*. In some cases this isn't a problem, such as with network security. For securing "data at rest", it's a big problem. It's not like we can point a merchant to the PCI specification and say "Do that." There are questions regarding which security technologies are *really* appropriate for credit card storage, but even more importantly, which implementation options merchants should employ. Our goal is to fill in the gaps with actionable advice for PCI compliance in the area of encrypted data storage, and provide specific recommendations for day-to-day credit card storage & management.

PCI-DSS says you must render stored Primary Account Numbers (i.e. the credit card numbers) unreadable — that much is clear. The specification points to several methods they feel are appropriate (hashing, encryption, and truncation), emphasizes the need for "strong" cryptography, and raises some operational issues with key storage and disk/database encryption. But that's where things fall apart — the technology, deployment models, and supporting systems offer hundreds of variations, and many of them are inappropriate for *any* situation. These nuggets of information are little more than reference points in a game of "connect the dots", without an orderly sequence or a good understanding of the picture you are supposedly drawing.

Our focus is to help you understand which specific security technologies and implementation models are appropriate, but first, let's get a clear picture of what needs to be accomplished. To meet the PCI security specification for credit card storage, the following objectives must be met:

- Protect the credit card number, expiration date, service code and card-holders name from logical or physical access.
- Use access controls to provide separation of duties between administrators and users who access credit card numbers.
- Securely store encryption keys, protecting them from exposure, unwanted replacement or misuse, and establish procedures to provide 'dual control' over key management.
- Log access and administration of key management and PAN data storage systems.
- Document your process and protection measures.

The Data Security Standard provides lots of technology options for secure data storage, and allows merchants wide latitude in meeting these requirements. In this report we will point you in the right direction with actionable advice on meeting the PCI encryption requirements.

Encryption Options

For secure data storage encryption is the most effective tool at our disposal. It safeguards data at rest and improves control over access. The PCI Data Security Standard specifies you must render the Primary Account Number (PAN - the credit card number) unreadable anywhere it is stored. We can hash, truncate, tokenize, or employ other forms of *irreversible* obfuscation, but we sometimes need to keep the original data and occasionally access it for payment remediation or auto-payment. Thus encryption or tokenization is usually the answer, and even when you use tokenization you still need to encrypt the original PAN data in the secure token database. There are at least a dozen different variations on file encryption, database encryption, and encryption at the application layer to choose from, making the decision as clear as mud. The following is a description of available encryption methods, with pros and cons of each.

What You Need to Know about Strong Ciphers

In layman's terms, a strong cipher is one you can't break. That means if you try to reverse the encryption process by guessing the decryption key — even if you used every computer you could get your hands on to help guess — you would not find it during your lifetime — or many lifetimes. The sun might implode before you got the answer, which is why we aren't picky about choosing one modern cipher over another. Plenty of them are considered 'strong' by the PCI Council and they provide a list for you in the PCI DSS Glossary of Terms (https://www.pcisecuritystandards.org/security_standards/glossary.shtml). Triple-DES, AES, Blowfish, Twofish, ElGamal, and RSA are all acceptable options.

Secret key ciphers such as AES must use a minimum key length of 128 bits, and public key algorithms (those which encrypt with a public key and decrypt with a private key, or vice-versa) require a minimum of 1024 bits. All the commercial encryption vendors offer at least these options, plus optional longer key lengths. You can choose longer keys if you wish, but in practical terms they don't add much security, and in rare cases they offer less. Yet another reason to not fuss over the cipher or key length too much as long as you stick with one of the well-vetted options included in most tools.

When you boil it down, cipher and key length are *far* less important than the deployment model. How you use encryption in your environment is the dominant factor for security, cost, and performance, so that's what we'll focus on.

Encryption Deployment Options

Merchant credit card processing systems can be as simple as a website plug-in, or they might be a geographically dispersed set of data processing systems with hundreds of machines performing dozens of business functions. Regardless of size and complexity, these systems store credit card information in files or databases; it's one or the other. And the data can be encrypted before it is stored (at the application layer), or when it is stored (file or database).

Database Encryption: This is the most common storage repository for credit card numbers. All relational databases offer encryption, usually as an add-on. Most databases offer both very granular encryption methods (*e.g.,* only on a specific column) or can encrypt the entire database. Because databases stores redundant copies of information in

recovery and audit logs, full database encryption is a popular choice for PCI to keep PAN data from accidentally being revealed. When you need granularity, the encryption functions can be invoked programmatically through a procedural interface, requiring changes to database queries that instruct the database to encrypt and decrypt as necessary. The database automatically alters the table structure to store the encrypted cipher output.

More commonly we see databases configured for *transparent* encryption; encryption is applied automatically to data before it is stored. In this model all encryption and key management happens behind the scenes without the user's knowledge or explicit application programming. It's easier to implement, but as we will discuss it can be slightly less secure and can't protect against administrator abuse.

File/Folder Encryption: Some applications, such as retail payment systems and some web applications, store credit card data within flat files. Encryption can be applied transparently by the operating system or additional software as files are written to disk. This type of encryption is offered as a third party add-on or embedded within the operating system. File/folder encryption can be applied to database files and directories as well, so the database contents are encrypted without any changes to the application or database. It's up to the local administrator to *properly* apply encryption to the right file/folder to ensure PAN data is not exposed.

Application Layer Encryption: Applications that process credit cards can encrypt data prior to storage. Whether file or relational database storage, the application encrypts data before it is saved, and decrypts before data is displayed. Supporting cryptographic libraries can be linked into the application, or provided in a third party package. The programmer has great flexibility in how to apply encryption, and more importantly can choose to decrypt on application *context*, not just user credentials. While all these operations are transparent to the application user, this is not transparent encryption because the application — and usually the supporting database — must be modified.

Application encryption is often built into off the shelf payment applications, obviating the need to deal with these (potentially) complex changes.

All these options protect stored information in the event of lost or stolen media, and need external key management services to secure keys and provide basic segregation of duties. We will go into much greater detail on how best to use each of these deployment models when we examine the use cases and selection criteria.

Procedural vs. Transparent Encryption

A quick note on transparent encryption, which has become an attractive choice for quickly getting stored credit card data encrypted. Traditionally encryption was performed manually. If you wanted a file or a database encrypted, it was up to the user to write a program that called the procedure or encryption interface (API) to encrypt or decrypt. Some databases and operating systems have add-on features that will automatically encrypt data prior to writing it to disk: this is transparent encryption. While procedural encryption offered fine-grained control and good separation of duties, it can be labor-intensive to implement in legacy systems. Transparent encryption requires little or no changes to the application or database code so is very easy to implement, but decryption occurs *automatically* for authorized accounts so security is no stronger than the (weakest) authorized password. The principal use case for transparent encryption is keeping media (*e.g.,* backup tapes) safe, but in some cases it is appropriate for PCI as well.

Selection Criteria

As a merchant your goal is to protect stored credit card numbers (PAN), as well as other card data such as card-holder name, service code, and expiration date. You need to protect all these fields from both unwanted physical (*e.g.,* disk, tape backup, USB) and logical (*e.g.,* database queries, file reads) inspection (but you are only required to encrypt the PAN). And if possible, detect and stop misuse as well.

Our goal is to offer pragmatic advice so you can accomplish those goals quickly and cost-effectively so we won't mince words. For PCI compliance, we only recommend one of two encryption choices: Transparent Database Encryption (TDE) or application layer encryption (Note that these recommendations also cover encrypted data in tokenization deployments, which is the only other data obfuscation option we recommend).

There are many reasons these are the best options. Both solutions offer protection from unwanted media inspection, with similar acquisition costs. Both offer good performance and support external key management services to provide separation of duties between local platform administrators, storage administrators, and database administrators. And provided you encrypt the entire database with TDE, both are good at preventing data leakage. Choosing which is appropriate for your requirements comes down to the applications you use, and how they are deployed within your IT environment. Here are some of the common reasons for choosing TDE:

Transparent Database Encryption

Time: If you are under pressure to get compliant quickly — perhaps because you can't see how you could possibly comply by your next audit. The key TDE services are very simple to set up, and enabling encryption is simple enough to roll out in an afternoon, assuming your database platform supports it.

Modifying Legacy Applications: Legacy applications are typically complex in function and design. Most do not consolidate communication with the database, rather they scatter thousands of queries throughout the application. To implement application layer encryption requires code changes everywhere data is written to storage, and is likely to cause effects and 'bugs'. Transparent encryption avoids modification to the application or database.

Application Sprawl: As with hub-and-spoke workflows and retail systems, you could easily have 20+ applications that all reference the same transaction database. Employing encryption within the central hub saves time and is far less likely to generate application errors. You must still mask output in the applications for users who are not entitled to view the credit card numbers and pay for that masking, but TDE deployment is still simpler and likely cheaper.

Application Layer Encryption

Transparent encryption is easier to deploy and its impact on the environment is more predictable, but it is also less secure and less flexible than employing encryption at the application layer. Given the choice, most people choose cheaper and less risky every time, but there are compelling arguments in favor of application layer encryption. The following are common decision points:

Web Applications: These often use multiple storage media, for relational and non-relational data. Encryption at the application layer allows data storage in files or databases — even to different databases and file types simultaneously. And it's just as easy to embed encryption in *new* applications as it is to implement TDE.

Access Control: Application layer encryption offers a much better opportunity to control access to PAN data because it inherently de-couples user privileges from encryption keys, and offers options for advanced access controls, biometrics or federated identity. The application can require additional credentials (for both user and service accounts) to access credit card information; this provides more control over access and reduces susceptibility to account hijacking.

Masking: The PCI specification requires masking PAN data displayed to those who are not authorized to see the complete account number. Application layer encryption is both better at determining who is properly authorized, and also at performing the masking itself. Most commercial masking technologies use a method called 'ETL' which *replaces* PAN data in the database, and complicates secure storage of the original PAN data. View-based masks in the database require an unencrypted copy of the PAN data, making the data accessible to DBAs.

Security: Application layer encryption potentially provides better security: there are fewer places where the data is unencrypted, fewer administrative access points, better access controls, more contextual information to determine misuse and one less possible platform (the database) to exploit. Application layer encryption allows for multiple keys to be used in parallel. While both solutions are subject to many of the same attacks, application layer encryption is more secure. The caveat is that we often see poor implementations of application level encryption which end up reducing security.

File/Folder Encryption

For PCI compliance there are a few use cases where we recommend file/folder-level encryption, transparent or otherwise. In cases where a smaller merchant is performing a PCI self assessment, file/folder-level encryption offers considerable flexibility. Merchants can encrypt at either the file or database level, which provides uniformity in organizations with multiple database types. Most merchants with mainframe and flat file databases employ file level encryption as it works with their architectural model and, in some cases, is the only option available.

Great care must be taken to ensure all relevant data is encrypted; audit logs, transaction logs and cached files contain sensitive information. *It's up to the platform administrator to correctly apply encryption to ensure data does not leak from one of these sources.* You will need to use external key management to meet the specification, as well as segregate administrative users from those who access card data. File encryption is not our first choice, as it is more difficult to provide separation of duties and leaves more room for mistakes and mishaps, but it meets the PCI requirements and delivers very good performance.

Supporting Systems and Features

Encryption relies on various supporting security systems and features to work effectively. Key management and access controls are important building blocks, and thus subject to audit to ensure compliance with the Data Security Standard.

Key Management

Key management considerations for PCI are pretty much the same as for any secure deployment: you need to protect encryption keys from unauthorized physical and logical access; and to the extent it's possible, prevent misuse. Per our introduction, we will avoid talking about ISO specifications, key bit lengths, key generation, and distribution requirements, because quite frankly you shouldn't care unless you're programming your encryption from scratch (a *really bad idea*). More precisely you should not *need to* care, because you pay commercial vendors to get these details right. PCI drives a large amount of encryption product sales, so the majority of tools evolved to meet PCI requirements.

What you want to consider is how the key management system fits within your organization and works with your systems. There are three basic deployment models for key management services; external software, external Hardware Security Module (HSM), and embedded within the application or database.

External Hardware: HSMs provide extraordinary physical security, and most are custom-designed to provide strong logical security as well. Most have undergone rigorous certifications, the details of which the vendors are happy to share with you because they take a lot of time and money to pass. HSMs offer very good performance and take care of key synchronization and distribution automatically. The downside is cost — this is by far the most expensive key management option. And for disaster recovery planning and failover, you're not just buying one of these devices, but several. They don't fit into virtual environments as well as software, which can run directly on the virtual machines. We have heard a handful of complaints that the APIs were difficult to use when integrating with custom applications, but this concern is mitigated by the fact that many off-the-shelf applications and database vendors provide the integration glue for you.

External Software: Another common option is software-based key management. These products are typically bundled with encryption software but there are standalone products as well. The advantages are reduced cost, compatibility with most commercial operating systems, and good performance in virtual environments. Most offer the same functions as their HSM counterparts, and offer reasonable performance given enough processing resources. The downside is that these services are easier to compromise, both physically and logically (but are still very secure). And it's your responsibility to provide the platform to run the software. Key server software benefits from being deployed on dedicated systems, and you must ensure that the platforms they're installed on are fully secure.

Embedded: Some key management offerings are embedded within application platforms: try to avoid these. For years database vendors offered database encryption but left the keys in the database. That meant not only DBAs had access to the keys, so did any attacker who successfully executed an injection attack, buffer overflow, or password guess.

Some legacy applications still rely on internal keys and they may be expensive to change, but this is mandatory to achieve compliance. If you are using database encryption or *any* kind of transparent encryption, make sure the keys are externally managed, even if said management is a feature of your embedded product. This makes it possible to enforce separation of duties, provide adequate logical security, and make it easier to detect misuse.

By design all external key management servers have the capacity to provide central key services, meaning all applications go to the same place to get keys. The PCI specification calls for limiting the number of places keys are stored to reduce exposure. You will need to find a comfortable middle ground that works for you. Too few key servers cause performance bottlenecks and poor failover response. Too many cause key synchronization issues, increased cost, and increased potential for exposure.

Over and above that, the key management service you select must provide several other features to comply with PCI:

Dual Control: To provide administrative separation of duties, master keys are not known by any one person; instead two or three people each possess a fragment of the key. No single administrator has the key, so some key operations require multiple administrators to participate. This deters fraud and reduces the chance of accidental disclosure. Your vendor should offer this feature.

Re-Keying/Key Rotation: Sometimes called key substitution, this is a method for swapping keys in case a key might be compromised. If a key is no longer trusted all associated data should be re-encrypted; and the key management system should have this facility built in to discover, decrypt, and re-encrypt. The PCI specification recommends key rotation once a year.

Key Identification: There are two considerations here. If keys are rotated, the key management system must have some method to identify which key was used. Many systems — both PCI-specific and general-purpose — employ key rotation on a regular basis, so they provide a means to identify which keys were used. Further, PCI requires that key management systems detect key substitutions.

Each of these features is required, and you will need to verify that they meet your expectations during an evaluation.

Access Control

Key management protects keys, but access control determines who gets to use them. The focus here is how best to deploy access control to support key management. A couple points of guidance in the PCI specification concerning use of decryption keys and access control settings frame the relevant discussion points:

First, the specification advises against using local operating system user accounts for determining who can have logical access to encrypted data *when using disk encryption*. This recommendation is *in contrast* to using "file — or column-level database encryption", meaning it's not a requirement for encrypting database contents. This is nonsense. In reality you should eschew local operating system access controls for *both* database and disk encryption. Both suffer from the same security issues, such as potential discrepancies in configuration, so local administrative roles should not be considered equivalent to domain administrative roles. Use domain access controls for both.

Section 3.4.1 of the specification is where most people get confused. The assertion that "Decryption keys must not be tied to user accounts" leaves a lot of room for interpretation, but if you carefully consider this statement it actually cuts to the heart of the matter. Some interpret this as meaning keys should not be tied to a single user account, but rather a *service* account specifically configured for sensitive data access. Most merchants regard this statement as nothing more than a redundant way of saying you need to set domain level access controls, placing the assertion in context with the

rest of Section 3.4. Still others feel this demands a separation of identity management and authorization, meaning the right to decrypt data is distinct from possession of account credentials.

We recommend you comply with all three interpretations:

Service Account: Use a service account that requires additional credentials over and above what normal users provide, such as a digital certificate. Using a service account is much easier for account management, and has the added benefit that auditing chores are easier when you can focus on a single account.

Domain-Level Identity Management: You need to use domain level credentials to avoid attacks predicated on misconfigured servers or inappropriate rights bestowed on local administrators.

Verify Authorization: Above what's provided by access control, verify authorization rights that take into account proper usage policies. For example, while domain access controls such as Active Directory and LDAP services may be used, applications and databases typically maintain authorization rights internally. They do not inherit *rights* from the domain — only identity. Instead databases provide extensive facilities to map their authorization rights to external accounts. Similarly, implementing encryption at the application layer has the inherent benefit of gating access based on business context: data is decrypted only when it makes sense in the context of the function being performed. This is a great way to detect misuse!

Auditing and Verification

Any system you choose should provide audit logs of all administrative activity, failed logins, and system failure. If you are a Tier 1 merchant you must provide your auditor with not only these logs, but also with specific reports on system setup. Your vendor should be able to provide the necessary reports for checking configuration, reviewing administrative access history, listing approved administrators, detailing system failures, and any other pertinent security information. They should provide documentation that discusses key management processes, as well as reports from third party analysts or security certifications. These reports, and the means to collect the audit data to populate them, should be built into the product.

Conclusion

The Payment Card Industry (PCI) Data Security Standard (DSS) was developed to encourage and enhance card-holder data security, but in order to succinctly address diverse audiences on a breadth of security topics, clarity and pragmatic guidance is lacking. Connecting the dots between the requirements and real world technologies that meet the requirements need not be as difficult as the specification would lead you to believe. Transparent encryption for databases and files, as well as embedding encryption into your applications, offer different advantages. Provided with supporting key management and access controls, these options meet the standard *and* provide good data security. All are commercially available to drop into existing environments, and can be implemented with minimal impact to operations. The common selection criteria presented should arm you with the information you need to make a sensible choice.

Who We Are

About the Author

Adrian Lane, Analyst and 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, Vice President of Engineering at Touchpoint, the CTO of the security and digital rights management firm Transactor/Brodia. Adrian also blogs for Dark Reading and is a regular contributor to Information Security Magazine. 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 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 evaluation, and merger and acquisition assessment.
- Investor consulting: technical due diligence, including product and market evaluation, available in combination with deep product assessment 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.