Protecting Stored Cardholder Data (An Unofficial Supplement to PCI DSS v3.0)

Rob Chahin — rob[dot]chahin[at]nccgroup[dot]com

iSEC Partners, Inc
123 Mission Street, Suite 1020
San Francisco, CA 94105
https://www.isecpartners.com

February 12, 2015

Abstract

In what started as a blog post that got too long, I take a look at PCI DSS v3.0 Requirement 3.4 - the requirement to protect cardholder data on disk / at rest. It is a security industry mantra that “there is a difference between compliance and security”, and the requirement to protect stored cardholder data is an excellent example of this. There are a number of compliant options available, with varying levels of security in different scenarios. This document is intended as an analysis of the various compliant options such that the reader can choose an option that makes sense - and in doing so, meet their compliance obligations while also improving security and keeping costs proportionate. Ultimately, all of the advice in this document boils down to two things: don’t store any data that you don’t need to, and always consider what you’re actually protecting your data from. Different methods of storage have different security properties, and what works for one organization might not work for another.

1 Version History

February 12, 2015 — v1.2: Minor clarification on the state of MD5 preimage vulnerabilities.

February 11, 2015 — v1.1: Thanks to Jann Horn for pointing out that the definitions of collision resistance and second preimage resistance had become reversed.

February 9, 2015 — v1.0

2 The Requirement

I was looking for some additional guidance or rigor in PCI DSS v3.0 regarding the ‘correct’ way to store cardholder data, and reading other blog posts and presentations on the subject tells me that I wasn’t alone. While PCI DSS v3.0 did add an optional salt to card number hashes, unfortunately nothing else changed. I had originally planned an extremely in-depth piece on the use of hashing, but the more assessments I do and the more approaches I see, I feel that a more general view is needed. All of the most important points on the use of cryptographic hashes are still included in the hashing section.

Doing some research before writing this document, I came across varying levels of accuracy and usefulness in the current guidance:

> "**One-way hash functions based on strong cryptography** — also called hashed index, which displays only index data that point to records in the database where sensitive data actually reside."

Source: `https://www.pcisecuritystandards.org/pdfs/pci_fs_data_storage.pdf` *(2008)*

Frankly, this is just incorrect, but it was written in 2008.

> "PANs stored in primary storage (databases, or flat files such as text files spreadsheets) as well as non-primary storage (backup, audit logs, exception or troubleshooting logs) must all be protected."

Source: PCI DSS v3.0 Requirement 3.4

Protecting the card number is hopefully evident to anyone subject to PCI DSS, but that's the point of every requirement. The guidance here falls short of articulating any risk.

> "Full disk encryption helps to protect data in the event of physical loss of a disk and therefore may be appropriate for portable devices that store cardholder data."

Source: PCI DSS v3.0 Requirement 3.4.1

It seems that the SSC is acknowledging a critical point here: that **full-disk encryption is valuable primarily for movable or removable media.** The requirements also acknowledge a difference between file, column and disk encryption, but don't articulate what that difference is.

The requirement is specified in DSS v3.0 with auxiliary information in information supplements and FAQs, all of which can be easy to lose track of when you're looking at hundreds of other requirements. The standards documentation does not currently provide guidance on when different kinds of storage are appropriate.

3 The Point

You can pick any one of the compliant data storage methods, implement it, and have your assessor sign off. However, with a handful of different approved methods of storage — all of which function very differently — **the more important question is: what are you protecting the cardholder data from?**

Each of these methods protects against different threat actors (the people trying to steal your data) and compromise scenarios in different ways, but this is often not considered when deciding which method to go for.

As it stands, these are your options:

- One way hashes, meeting the definition of "Strong Cryptography"; with an optional but "not required" salt, specifically to protect against pre-computed hashes.
- Truncation, defined as "permanently removing a segment of PAN data", most often by deleting all but the first six and last four digits.
- "Tokens and one-time pads" — note that these are two distinct things, and are explained separately in this document.
- Encryption plus good key management.

Let's take a look at some pros and cons. Some vendors conveniently forget to include the cons in their own documentation.

4 Types of Cardholder Data Protection

4.1 Truncation

This is the easiest option to address. If you don't need the card number — for example, you process a transaction as soon as you receive a card number, and you don't need it for repeat transactions — you don't keep it. **The first six and last four digits are not protected under PCI**, and are mostly useless to an attacker. You generally do not need to protect these for compliance purposes, but be mindful of who has access to them, and what other data they might have access to. There was a high-profile breach of an email service in 2012 because the provider was using the last four digits as identity verification.

First six and last four are sometimes an alternative for hashing, as — combined with other cardholder data — they can uniquely identify a card, if not across your entire customer base, at least within the range of people with a given name. They're not guaranteed to be unique, so another method is necessary if uniqueness needs to be guaranteed.

The last four alone are often stored in clear text so they can be displayed to a customer to help them identify their own cards. Stored alongside an encrypted or tokenized number, this is often not a problem, but **storing cleartext alongside a hash is a major problem**. This is discussed further in the hashing section.

4.2 One-Time Pads

Popularized in the two World Wars, one-time pads are a mechanism for perfect encryption. By combining your secret message (card numbers) with a secret value (the pad), the encryption is unbreakable for as long as the key is unknown.

If you're wondering why we don't use this for encrypting everything, it's because it has some major limitations. First, the pad has to be as long as the data you're encrypting (hence 'pad'), and can never be reused (hence 'one-time'). For a 16-digit credit card number that's easy enough, but for large files it gets tricky — for one thousand 1MB photos, you'd need a 1GB encryption key. Second, you need a way of securely transmitting the secret to someone else and a secure location to store it — if you have those, then why not just use those to store and transmit the card number?

One-time pads still see a lot of use in the military, but not so much in commerce. You can think of it as encryption with gigantic encryption keys and no built-in key exchange mechanism. Some of the other options in this list use it as part of a larger solution.

4.3 One-way hashes

4.3.1 Why MD5 is really broken

It's pretty widely known that MD5 is 'cryptographically broken', and does not meet the PCI requirements for strong cryptography. However, it's widely misunderstood why this is.

One-way hashes are great. They have a wide range of uses, from storing a secret value in a way that lets you match input to a secret without knowing what the secret is, to providing digests of large data for integrity checking. They have so many uses because they have so many properties, but for storing card numbers we rely only on one of those.

The property that we rely on is called 'preimage resistance', and it means that given the hash of a card number, you have no way of calculating the original card number, i.e. the hash is truly one-way. If you lose all of your card hashes, it should be impossible for an attacker to derive the card numbers.

Two of the other big properties of a hash are second preimage resistance, and collision resistance.

Second preimage resistance means that given a card number, you cannot find another card number with the same hash. This is good when we store a card number because we're usually checking a card against a hash to see if it matches. If there were collisions, we might get a false positive match. Fortunately, credit card numbers have a limited range of values, and we can pretty trivially demonstrate that for ALL card numbers, there are no collisions with any hash function in use today.

Collision resistance is like a combination of preimage resistance and second preimage resistance; it means that you cannot pick two card numbers that result in the same hash, without the limitation of having to start with a given number; you get to choose both. This property is made redundant in the same way as second preimage resistance.

So for storing a hash of a card number, we really just need preimage resistance. If we take a look at MD5 as the example of the weakest hash still widely used, it has that for our purposes. The broken part is the collision resistance — you can generate two messages with the same MD5 hash, but not when your messages are 16-digits. We know that because we can easily generate all of the possible 16-digit MD5 hashes.

That last statement is critical. **The same thing that allows us to generate all MD5 hashes to demonstrate that collision and second preimage resistance are still in place for credit card numbers is the same thing that makes MD5 unfit for use — it's really fast.**

4.3.2 Why the alternatives are broken as well

According to the oclHashcat website[1] (a tool for bruteforce attacks on hashes) and some benchmarks generated on real-world hardware, we can generate 8.5 billion MD5 hashes per second on a computer with a $200 graphics card. Build a machine designed to bruteforce hashes, and we can pretty easily reach 100 billion MD5 hashes per second.

An AMEX PAN is 15 digits long, and the last digit is predictable when you're guessing because it's a Luhn digit. An AMEX also starts with either a 35 or a 37, so the first digit is known, and the second digit is easily guessed. That leaves us with 2 trillion possible combinations.[2]

On our very high-end test machine above, it would take 20 seconds to try all of them. (There are some minor complications — generating the Luhn digit isn't a feature of the tool, so that becomes limited by the CPU. In practice we would actually just add the last digit as another unknown digit — so it would take 200 seconds to try all of them.)

No QSA worth his or her salt (no pun intended) will have signed off MD5 storage in the history of PCI, and that's great news — the cryptographic weaknesses in MD5 have been known for a long time, and every practicing QSA should know this. This weakness is the reason that MD5 doesn't meet the PCI definition of 'Strong Cryptography'. In fact, the PCI definition of 'Strong Cryptography' doesn't even include hashing algorithms; it defers to NIST SP 800-57 Part 1, which says that the SHA family of hashes are allowed.

SHA-1 on the same machine we use as an example above can be bruteforced at a rate of about 30 billion hashes per second. That's 30% of the speed of MD5, so even factoring in the extra work to include the last digit in our bruteforce attack, that takes us to 11 minutes.

Similarly, for other popular hashes:

SHA-256: 27 minutes

SHA-512: 73 minutes

This type of attack is highly parallelized — in the examples above, we can actually test the card numbers against every single hash you store at the same time by generating a candidate hash once, then comparing it extremely

[1] `http://hashcat.net/oclhashcat/`

[2] Some papers on this topic also assume that the first six digits — the issuer or bank identifier number (IIN/BIN) — are predictable. This would only be true in edge cases where you store detailed information about the type of card that a customer is using, which is rarely done in practice. However, a list of most IIN/BIN codes is not difficult to acquire, so if you do store additional card metadata such as this, be aware that your hashes are weakened further.

quickly to every stored hash. Our 73 minutes figure means we can 'crack' every one of your ten million stored cards in 73 minutes (give or take a small overhead)..

If you consider MD5 broken for the purposes of storing a card number, you should also consider all of the SHA hashes as broken as well.

4.3.3 What ISN'T broken?

There are two techniques used to prevent the bruteforce of hashes described above — salting and slow hashing.

First, salting. In PCI DSS, this is the (optional) "additional, random input value" described in the guidance column. By introducing this random value, you prevent pre-generation of all possible hash values, and more importantly, you prevent the parallelization of the attack above. If every card uses a unique salt, the attacker has to attack each card individually — so we would need to spend up to 73 minutes attacking the first card, 73 minutes attacking the second, and so on.

Even with a salted SHA-512 hash though, we're losing a credit card on average every 37 minutes (because, statistically, we will find a card halfway through the search). That's not good enough. The situation gets worse when you look at why people are using salts. Often, it's to store a list of all of the cards you've received, and to check a new incoming card to see if you already have it — for example, if you're doing your own anti-fraud or blacklisting. However, to compare that new card against your stored cards, you need to know the salt of the card you're comparing it to. You don't know which card you're comparing it to in a blacklist scenario, so you have to generate one hash of the new card for every unique salt you already have. You are effectively attacking your own card numbers. To work around this, people use a shared salt, or no salt at all, and remove the primary benefit of having a salt in the first place.

Another point of confusion is the concept of a secret salt. Rather than storing the salt with the hash (which is the correct place to store it), some companies store the salt away from the hash, using the logic that an attacker stealing the hashes won't have access to the salts, and therefore won't be able to bruteforce the hashes. If an attacker has compromised your database, why assume that they haven't also compromised your secret salt location? The salt is not an encryption key, but if you treat it like one, expect to be assessed against the key management requirements. Storing the salt elsewhere introduces complexity, but rarely security.

The second technique is using a hash that isn't designed for speed. "Slow" hashes like bcrypt and scrypt are designed to require a lot of work, and to be scalable to require much more work as our computers get faster.

Attacking bcrypt hashes with our test machine results in around 1,700 hash candidates per second — over two million times slower than attacking SHA-512. This is using a work factor of 10 — bcrypt is configurable, so you can raise or lower this number as necessary. Work factor 10 is a reasonable medium when considering that new credit card numbers are accepted relatively infrequently. With hashes generated at 0.0000005% the speed, the hashes are defeated at a much more reasonable pace — about every 50,000 days, on average. This is assuming unique salts, which are built into bcrypt. If you force the hash to use a shared salt because you're doing blind lookups, then it will take 100,000 days to find every single card number you have stored. If that's one million cards, expect to lose ten per day.

The major limitation to using slow hashes, or any type of hash, is that a bruteforce attack generates hashes just as you would. In fact, oclHashcat has a number of shortcuts that allow it to generate and test them faster than you can,[3] and your attackers can use specialized hardware, while you're limited to general-purpose compute resources. The more work you require for the hash, the slower they can be attacked, but the slower they can be generated by your own servers. **No matter how much work you require, hashes can still be attacked.** As it stands, hashing is an arms race, limited by how much work they require (based on your choice of algorithm), and how quickly your attacker can generate them (based on their computing resources and more efficient tools).

[3] http://hashcat.net/events/p13/js-ocohaaaa.pdf

If a hash is absolutely required — and often it isn't — the goal is to make it unattractive to attack them, and to buy cardholders time to replace their cards before they are defrauded. **If you lose those hashes, you've still been breached.**

4.3.4 Storing hashes with cleartext

One thing that PCI DSS v3.0 is clear about is the increased risk if partial cleartext is stored alongside a hash. Using the numbers above, if you store the last four digits in cleartext, you reduce the security of your hash by a factor of 10,000. Attacks that would have been measured in hours-per-card will now be measured in cards-per-second. If you store the first 6 and last 4, you may as well give up.

If you currently store, or are planning to store, partial cleartext alongside a one-way hash, model how long it would take to perform a brute force attack. No matter how strong your hash is you'll likely find that it's not strong enough.

4.4 Encryption

If you have a requirement to store the card numbers locally and to retrieve them at a later date, perhaps to send them on to third parties, encryption is the way to go. However, there are many types of encryption that protect against different things, and **your choice of encryption method should be informed by the threat actor**. Let's look at the common places to encrypt: in the application, in the database, and on the disk.

In the first of these, your application encrypts data as it's received. Everything downstream of your application has no access to the keys, so the data is protected from attacks against those systems. If the application accepting card data doesn't need to see the card data at a later date, you'll use asymmetric encryption — the application will encrypt with a public key, and then even the application will be unable to decrypt the card data. The private key will be stored in another, much more restricted part of your network, and all card data decryption will take place there. Using an e-commerce website as an example, this would protect against many common website attacks — assuming that the website infrastructure and the decryption environment are suitably separated.

This is probably the best place you can encrypt data, but there's a major caveat. Somewhere, there is a key to decrypt the data. **Even if you store the key in an HSM, there is some application that has access to decrypt data**; if there weren't, then why is the data using reversible encryption? You can massively limit the exposure of the key or decrypting application, and you can apply additional controls like rate-limiting and other context-dependent authorization, but it's not bulletproof. I've had clients try to work around this by telling me that the key is encrypted — but where is the key-encrypting key? And if that's encrypted, where is the key that encrypted THAT one? To paraphrase Terry Pratchett, it's keys all the way down. Note that key management is a problem regardless of which encryption method you use; but the other forms of encryption have bigger problems that overshadow this one.

A layer down, and much easier to implement, is database encryption — whether that's tablespace encryption or column encryption. In this model, the database manages the keys and encrypts all sensitive data it receives. It's encrypted on disk and only accessible through the database interface (e.g. just reading the database files or filesystem backups won't disclose data). However, it IS still accessible through the database interface. **Anyone with access to query the data in the database can query the cardholder data in cleartext.** You can use access controls to limit this, but then you're relying on the access controls, not the encryption. This protects against an attacker with access to the database storage location but no access to the database itself, but does nothing to protect against SQL injection. Column-level encryption can also be used to hide data from your database administrators, but practically, a motivated and malicious DBA will be able to work around this restriction — after all, they have access to the database settings and all of the user accounts.

Further down still is disk encryption. You can encrypt the entire disk so that any data written to it is automatically encrypted. While this provides extremely convenient encryption, it also provides convenient decryption. Anyone with access to the system can also read the decrypted data. Full disk encryption only really protects against the physical theft or loss of the machine or drive — which is great for a USB drive or a laptop, but doesn't add much to the servers in a secure datacenter, and **does nothing to protect against vulnerabilities in your software.**

So how do you choose which layer to encrypt your data at? PCI DSS doesn't yet provide guidance for making that decision — requirement 3.5 clearly allows for full disk encryption, so if compliance is your goal, why not go with that? Well, because it's expensive and operationally complex.

In addition to specifying encryption requirements, the DSS specifies key management requirements. Ultimately, there must be some key that isn't stored on the server. That means it must be stored by a separate service (like an HSM), or by people (as components). Having people act as key custodians means that those people need to be available when your server restarts, or it will be unable to read its own data. For most organizations, that option is ruled out. There are plenty of HSMs and key management appliances available that will allow your servers to boot and decrypt themselves without a human (as long as the HSM itself doesn't need to restart), but they're frequently expensive.

So we've established that full disk encryption, and to some extent database encryption, are expensive, awkward, and don't defend against much. That means we can do one of three things:

1) Leave everything as it is, and require expensive, awkward encryption that protects from a very small set of attacks.

2) Require encryption at a higher layer than disk or databases. This can add pretty great security, but it's also more complex, and requires a decent understanding of cryptography from your application developers or vendors.

3) Forego the requirement for encryption for data that is physically protected.

Where this goes in the future will really come down to which attacks the SSC and the card brands are trying to prevent with requirement 3.4. If the risk is physical theft, then hopefully the requirement will be clarified to state that only media that could be feasibly stolen needs to be encrypted, and the SAN in your locked cage in a locked datacenter with an armed guard doesn't need to be backed up by $30,000 of crypto hardware.

From a security perspective, if you're going to use encryption, you need to be aware of what your threat model is. **When you decide to encrypt data, think about how it's going to be decrypted — and who is going to have the ability to do it.** It is a common fallacy that encrypted and protected are the same thing.

4.5 Tokenization

Much noise has been made about tokenization. It's arguably the newest approach to protecting stored data, and numerous vendors have popped up to take advantage of this new market. Like any other new trend, there's a lot of conflicting and often inaccurate information.

Broadly speaking, there are two models of tokenization:

1. Tokens sent to you via a third party such as a payment processor
2. In-house tokenization.

In the first model, a third party is responsible for storing the card number, and they give you a token to represent that card. Ideally, it's not actually generated from the card number directly, but is a reference (like an index). That index will be tied to that specific processor and to you, so the loss of that token is limited in how it could be abused.

This is a very effective control in scenarios like payment redirects and hosted checkouts. You never need to see the card number, so you never do, and the compliance obligations are shifted to the payment processor. They give you a token that you can use to reference the card number, both internally and when processing transactions, and you don't need to worry (much) about losing it.

In the second model, you develop, or usually buy, an application that converts card numbers into tokens. Some vendors are claiming that this provides huge scope reductions because tokens aren't in scope for PCI DSS and their appliance is hardened so it can't be broken into. Be wary of these.

There is no PCI DSS certification for a hardened appliance. If a vendor's hardware or software is installed in your environment, your QSA is going to audit it. Frequently, because these appliances are locked down, they are also unable to receive patches from your centralized management, and they may not have been hardened to your standards. Ask your vendor what ability you have to maintain the appliance, and then bear in mind that having to manage the security of an appliance is no less onerous during an audit than doing so for a server.

As for scope reduction, you need to understand how the tokenization product generates tokens. For a third party model, this is irrelevant — their auditor will check this during their assessment. For in-house tokenization, the tokens are either an index value of a card number stored on the appliance, or they're an encrypted value using some proprietary encryption or obfuscation scheme. Either way, you need to understand how the PCI requirements in section 3 are being met. If the appliance performs encryption of any sort, you or your vendor will need to demonstrate to your auditor how the encryption is performed, how it's secure, and how the keys are managed.

Another problem with in-house tokenization is that a token that can be converted to a card number is still treated as a card number, until it moves to an environment where it cannot be converted to a card number. Again, for a third party token provider, this is fine — just make sure the third party doesn't give you a mechanism for retrieving PAN. When it's in-house, though, usually the appliance will give you a mechanism for requesting PAN. Any user or application that can use this mechanism is in scope for your audit, and represents a security risk.

To re-iterate that point: **if your tokens can be converted back to credit card numbers, it doesn't matter how they were generated or who generated them — they are back in scope to some extent, and so are the systems that store and process them.** They might not all be in scope for applying all controls, but they will be in scope for assessing which controls should apply.

One of the great uses for in-house tokenization is a hub/spoke model. If you have a central datacenter that provides services to dozens of branch offices, you may be able to keep the branches out of scope by only sending them tokens while keeping the PAN protected in the datacenter. Just bear in mind that this is functionally not different to encryption — it's just sold to you in a neatly packaged bundle that will hopefully already have its security proven, but don't assume that it has. **The device that performs tokenization will always be in scope for assessment**, unless the SSC releases a special certification.

5 Conclusion

There are a number of ways to meet requirement 3.4, but some of them are poorly understood. The most important consideration is not what is the easiest, but what you are defending against.

A reminder of your options:

- **If you don't need card data, don't store card data. You'll save time and money, and you'll sleep better.**
- Hash the data if you need to compare two cards to each other, but question whether you really need to, and remember that it will always be vulnerable.
- If you're relying on a secret value, you're using encryption, and you'll need to meet the key management requirements.
- If you do encrypt cards, think about where the encryption and decryption takes place, and how that does or does not defend against various threats.
- Tokenization performed by someone else is great. Tokenization performed in-house is great if you have a distributed infrastructure. Be wary of vendors with the Next Big Thing, and have your engineers or your QSA speak to them to figure out what the "tokenization" process actually is. It should not be a secret.

6 Acknowledgement

Thanks to Jeremi Gosney of Sagitta HPC, who not only ran some hashing benchmarks for me on one of their hash-cracking machines (a Sagitta Brutalis, with 8 x GTX 980 cards), but has for some time been answering my questions on hash-cracking and letting me run my own tests on some of their other hardware. This document would be significantly less accurate without his input.