Title

Encryption of cardholder information

Doc. ID	Version
NFT-TA-09-J1	A
Produced by	**Date**
Torbjörn Lofterud	2011-04-26

1 Executive Summary

The PCI DSS standard requires encryption or hashing as ways to protect cardholder information, specifically the card number, the **P**rimary **A**ccount **N**umber (PAN).

While working with PCI DSS customers and from forensic investigations its quite clear that the techniques for how to apply strong cryptography to cardholder information isn't explicitly explained by the standard and therefore the encryption and hashing requirements are interpreted differently.

For example, in discussions with customers Cybercom has brought up the issue of the risks introduced by combining truncated PAN's together with hashed PAN's. PCI DSS 2.0 addresses this situation with a short note that "*It is a relatively trivial effort for a malicious individual to reconstruct original PAN data if they have access to both the truncated and hashed version of a PAN".*

There are at least three (3) different scenarios where cardholder information appears to be protected in compliance with the standard but still vulnerable if disclosed.

1) Cardholder information is hashed using a one-way-function such as SHA1 or MD5 without proper salting.
2) Cardholder information is encrypted using 3DES symmetric encryption in electronic codebook format (ECB).
3) Cardholder information is encrypted using public key cryptography without random padding.

This paper the describes attacks for each of the scenarios in order to provide guidance for how to implement strong encryption and avoid common pitfalls.

2 PCI DSS encryption requirements

The primary objective of the Payment Card Industry Data Protection Standard (PCI DSS) is to safeguard cardholder information such as the **P**rimary **A**ccount **N**umber (PAN) and the sensitive authentication data (CVV2, Track 1 and 2).

Chapter 3.4 deals with the details regarding encryption and key management.

> 3.4 *Render PAN unreadable anywhere it is stored (including on portable digital media, backup media, and in logs) by using any of the following approaches:*
>
> - ***One-way hashes based on strong cryptography***
> - *Truncation*
> - *Index tokens and pads*
> - ***Strong cryptography** with associated key-management processes and procedures*
>
> ***Note:** It is a relatively trivial effort for a malicious individual to reconstruct original PAN data if they have access to both the truncated and hashed version of a PAN. Where hashed and truncated versions of the same PAN are present in an entity's environment, additional controls should be in place to ensure that the hashed and truncated versions cannot be correlated to reconstruct the original PAN.*

What constitutes strong cryptography is further detailed in the glossary and in the PCI SSC FAQ documents as well as in periodic communication to security assessors. But one important factor is missing from the communication, the modes of operation for the cryptographic primitives.

The glossary specifically mentions AES, 3DES, RSA, ECC, Elgamal and SHA1 as "*industry-tested and accepted standards and algorithms for encryption*" but fails to address important issues such as RSA padding and cipher block chaining for 3DES and AES.

The glossary also recommends salting hashed card numbers, but its not a formal requirement. The PCI SSC FAQ states that it is "*a recommended practice, but not specified requirement, that a salt be included"*.

The requirements are quite clear on the fact that encryption and hashing needs to be implemented properly, but gives little guidance to developers or assessors as to what *strong cryptography* actually means.

3 Sample dataset

3.1 Databases

For this threat assessment three different databases have been created representing three different customer sizes. The first database has ten thousand cards and represents a smaller merchant. The second database has one hundred thousand cards representing a medium size merchant. The third database contains one million cards and represents a large merchant or a small payment service provider. Each database is populated with randomly generated PANs from the IIN distribution list and has proper checksums.

3.1.1 Issuer identification numbers

Each card number starts with an issuer identification number (IIN). The first six (6) digits of the PAN is used to identify the issuing bank. IINs are considerted secrets, but not a very well kept one, lists of known IINS can be found on the internet. For instance its quite easy to find lists containing the IIN:s used by most major swedish banks.

3.1.2 IIN distribution

The distribution of brands in the database has been created to reflect a small market, such as the Swedish one, with only a few larger banks dominating the market and a large selection of smaller brands.

IIN	Brand	Market penetration
910001	Bank A	25%
920002	Bank B	14%
930003	Bank C	7%
940004	Bank D	7%
950001 – 950021	Bank E-Z	47%

4 Hashing of PANs

4.1 SHA-1

The SHA-1 algorithm is a one way hash function. Each PAN is feed in clear text to the SHA1 function and the resulting 160bit hash is stored in hexadecimal form in the database. The only (known) way to attack SHA1 is through an exhaustive brute force attack.

4.2 Database contents

The sample database schema used for the attack against sha1 hashed PAN contains a transaction id, the hash of the PAN, the expire date and a customers ID. In order to be compliant with PCI DSS 3.4 the truncated PAN is not stored in the database.

4.2.1 Database scheme

Trans-ID	PAN (SHA-1)	Expires	CustomerID
1	E0b18e3d978fc06354f610bcb4f40915c6ab59f4	09/09	1
2	70da34d2c94426a5858165e0f24b6fea6ebc86e9	09/08	2
3	8b19dc6f5b7ef58b7f135dbc0492e0df6d1ff304	09/07	3

4.3 Attacking card numbers hashed with SHA-1

4.3.1 Partial known plaintext attack

Normally a brute force attach on SHA1 would be a daunting task considering the length of the input data (13 – 19 bytes). Most card numbers are 16 digits and these examples will focus on 16 digit cards.

Known IIN

The first digit of a real payment card number is always 4 or 5 followed by five bank specific digits, together they make up the 6 digit issuer identification number. Since lists of known IINs can be downloaded from the Internet the first 6 digits of the card number can be considered known plaintext.

Using this information the unknown information is brought down from 16 bytes to 10 bytes for a specific IIN. The number of different IINs in a small country such as Sweden is very limited.

Checksum (Luhn-10)

The last digit in the PAN is calculated based on the first 15 and can therefore simply be calculated as part of the brute force attack and thereby reducing the unknown by one additional byte down to 9 in total.

Exhaustive attack on a specific IIN

Since PANs are very predictable, the bruteforce attack can be reduced from 16 unknown bytes to roughly 9 unknowns for each IIN attacked. The conclusion is that 10^9 (1 billion) SHA-1 computations has do be done for each IIN.

4.3.2 John the ripper

John the ripper is a popular password brute forcing tool. It is widely known as one of the best password brute forcers available.

Patches for SHA-1

The default version of john the ripper doesn't come with support for raw SHA-1 hashes. It is however easy to find several different patches as well as pre-patched version of john the ripper that supports SHA-1 as well as other hash-algorithms such as md5.

File format

John the ripper normally loads password files. John the ripper expects the input as username:hash. Therefore the input needs to be modified to suit john the ripper but it is perfectly all right to have the same username for all hashes. During our tests we have used the format 'card:*sha1hash*'.

Incremental mode

John the ripper supports a brute force mode called 'incremental' in which it will simply try every single combination within the given parameters. For 16 digit PANs there are a large number of combinations to test. The incremental mode doesn't take into account the limited range of the first digit and the checksum and it is therefore very time consuming.

Using john the ripper to crack SHA-1 hashed pans in incremental mode is possible but slow. It is clearly unfeasible to crack any considerable amount of PANs from any of the three test databases within a meaningful timeframe. It does however provide a viable but slow way to find unknown IINs to use for further attacks.

4.3.3 Word list generation

John the ripper supports reading word lists from file or from standard input instead of using the incremental mode. In order to speed up the process a small and fast C-program called wordlist.c was developed that generate all of the one billion combinations for a specific IIN. The output of this program can then be feed straight into john the ripper.

```
./wordlist 950001 | ./john --stdin hashlist10k.txt
Loaded 10000 password hashes with no different salts (Raw SHA1)
```

Creating a specific tool to generate possible PANs from a specific IIN and to also only generate PANs with a valid checksum is clearly superior to the incremental attack. The attack scales linearly and the results are primarily based on access to good statistics on IIN usage.

4.3.4 Dataset speed considerations

It is clearly noticeable that the size of the database effect the speed of the attack but the scalability of John the ripper is truly remarkable. The amount of SHA-1 computations needed for each database is constant (10^9), it's the memory footprint and finding a match for each hash that takes considerably longer as the database crows. Even though the scalability is remarkable, a better search algorithm could probably increase the speed for larger datasets.

DB Size	Algorithm	Time / IIN
10k	SHA-1	8 minutes
100k	SHA-1	47 minutes
1M	SHA-1	525 minutes

For a database with 10 thousand PANs all of the PANs for a specific IIN can be recovered in 8 minutes.

4.3.5 Salting isn't enough

Using a global salt that is pre-pended to each PAN before hashing is only effective if the salt is kept secret. If the attacker knows the salt it can simply be added to the output from the wordlist-generation application. The speed of the attack is not affected by salting since this is a raw brute force attack and requires no pre-calculations such as rainbow tables. The only way to slow down the attack using a salt is to use a unique salt for each PAN.

4.4 Possible solutions

4.4.1 Don't use hashing as a security measure

PCI DSS suggests that hashing of PANs can be used instead of masking or encryption. Its obvious that hashing of PANs is far weaker than proper encryption and clearly inferior to truncation.

4.4.2 Slower hash algorithm

In order to slow down the brute force attack to a rate that makes it virtually unusable the algorithm needs to be at least 10 000 times slower than SHA-1. Using an algorithm that is 100 000 times slower might be advisable in order to secure against future improvements in algorithms and hardware. PBKDF2-HMAC-SHA1 provides a standardized and PCI DSS compliant alternative to SHA1 with configurable number of iterations.

4.4.3 Unique Salt for each hash

Using a unique salt for each PAN in the database slows down the attack significantly but also renders the hash unusable as a database key to use for lookups.

4.4.4 HMAC

One possible solution is to use an HMAC instead of a simple hash. In the HMAC a shared secret is used together with the PAN when hashing and thus making the attack impossible without the shared secret. Naturally the shared secret needs to be protected in a similar manner as any other type of shared secret or key used for encryption.

5 Symmetric encryption in ECB-mode

5.1 3DES-EDE-ECB

The PAN is encrypted using 3DES in EDE-ECB-mode. 3DES uses 64bit (8byte) blocks and the stored crypto text blocks have been divided with a colon (:) to ease readability. The 3DES key is kept secret and is assumed to be completely random.

5.2 Database contents

The sample database schema used for the attack against 3DES in ECB-mode contains a transaction id, the encrypted PAN, and a copy of the truncated PAN. Storing the truncated PAN in the database together with the encrypted PAN is quite common and is acceptable under PCI DSS 1.2 and 2.0.

The database has been populated with one million (1M) cards. The PANs has been randomly generated using the IIN distribution list from 3.1.2, the PANs have proper checksums. Each database-entry has been encrypted with 3DES in ECB-mode using a key that for this exercise is unknown. For convenience a truncated version of the PAN is also stored in the database.

5.2.1 Database scheme

ID	Encrypted PAN (3DES-EDE-ECB)	Truncated PAN
1	52b62426fb218e20:077d02e0e7c0b9b1	910001xxxxxx9377
2	68319cffbd7a08a1:f3247ac60cb4124d	950021xxxxxx9370
3	17fed566f5578160:96e17f2a4fdb9569	910001xxxxxx3163
4	318de67deb17f32a:e8e1a9416342b185	930003xxxxxx6889
5	52b62426fb218e20:2141378868234976	910001xxxxxx2839
6	bbe72f0cd8a91665:72b8f67694727019	910001xxxxxx7626

5.3 The attack

5.4 Attacking card numbers encrypted with 3DES-ECB

The heart of the problems is the fact that 3DES uses 64bit blocks and that a 16byte card number fits snuggly in two blocks. This divides the clear text in two and generates two separate cipher texts for each card number. The two halves of the PAN are encrypted separately. Different parts of the clear text are known due to the availability of the truncated card number. In this case the truncated card number is stored in the database, even if the truncated card number isn't available directly from the database it can often be found in log files or debug output.

5.4.1 Cipher block and clear text correlations

The first thing to look for is correlations between the first and the second cipher block. Every time there is a match some parts of the clear text can be easily recovered.

For example:

ID	Encrypted PAN (3DES-EDE-ECB)	Truncated PAN
1	d08a47587775c53d:**76b7617a1def7f8a**	910001xx**xxxx0107**
2	**76b7617a1def7f8a**:e3846771609583a	**910001xx**xxxx3582

In this case we can quickly draw the conclusion that the crypto text **76b7617a1def7f8a** corresponds to 91000107 since we have two different parts of the truncated card number to compare it with. Using this we can fill out some of the blanks in the truncated PANs:

910001xx91000107

91000107xxxx3582

Now one of the PANs (910001xx91000107) only lacks 2 digits, a total of 100 combinations, but we can also draw the conclusion that the missing digits is not '07' since that would have resulted in a completely duplicated crypto text. When we apply the LUHN-formula it reduces the scope even further, from 100 to 10.

```
9100010191000107 9100011991000107 9100012791000107
9100013591000107 9100014391000107 9100015091000107
9100016891000107 9100017691000107 9100018491000107
9100019291000107
```

If we can find a corresponding attack on the d08a47587775c53d–cipher-block we would have the complete answer, otherwise we would have to verify the remaining digits by trial and error at a friendly local web shop.

5.5 Possible solutions

5.5.1 PKCS#5

The PKCS standard #5 - *Password-Based Encryption Standard* provides guidance for how to properly salt and pad the message before encryption and to chain the resulting cipher text messages to avoid this and other threats.

5.5.2 Cipher block chaining

When cipher block chaining is used the second cipher block is encrypted using information from the first cipher block. This eradicates the problem with cipher block correlations between different blocks.

5.5.3 Ciphers with larger block size

Ciphers such as AES uses 128bit blocks instead of 64 bit blocks. When AES is used the entire PAN fits in a single cipher block and the correlation problems are avoided.

6 RSA with fixed padding

6.1 RSA public key encryption

An RSA key pair has been generated and for the purpose of this exercise the private key has been stored on an USB-stick, the USB-stick has been put in a safe and then the safe has been sunk at sea at an undisclosed location.

For each transaction the PAN is encrypted using the public key and then stored in the database together with a truncated version of the PAN that can be displayed to customer services and on the website when a customer performs a transaction. The public key is available to anyone with access to the system.

When encrypting with RSA the clear text data must be of equal length to the key, in this case 256 bytes (2048 bits). A 16-byte card number must be padded with additional data so that the total length is 256-bytes; in our sample the data is simply zero-padded.

9100016891000107 is turned into:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
000000000009100016891000107
```

The padded data is then encrypted using the 2048 bit public-key and the result base64 encoded for easy storage in the database.

```
EUNpuFS8ZdcbWFPjsAHuJt8wRr+8v6IFVaGrniuB0l6DQAYDRWIWjd/0X
HIPPh72+KnwCc1yaIWPDPthu+BLHboy48kCUTBnU0cGPjRmCSLsHB1Qrf
NAUxuMyj3YsnvSQe3kekA/GZSK7UBR2CwOIQNsBiS58oT470aKCj+XLuZ
h9zxod41INBzqUKt7v64TZnE6BB9zm3BQVwxvqj9BHWX3HPDrJ8lWJh7u
JVtBtv2wjhl040rwIRxuwh6Y0xYzx+5CVra3APXCG1vsVQxbCCnT1BTjC
FCtch1WddYZ7Iy2LFLpa+moCUb4drcDNSNTmAKM3oUsd5Oc5UwpceWDxQ
==
```

6.2 Database contents

For the third threat assessment the database has been populated with one million (1M) cards. The PANs have been randomly generated using the IIN distribution list in 3.1. and have proper checksums. Each database-entry has been encrypted using RSA with a 2048 bit public key.

6.2.1 Database scheme

ID	Encrypted PAN (RSA 2048)	Truncated PAN
1	EUNpuFS8ZdcbWF…Oc5UwpceWDxQ==	910001xxxxxx0107
2	RqbTx9HpiJ/y4o…oltekI7NKWFg==	950020xxxxxx2538
3	jsbn2XovfX9TCE…rE0RBflW6c+g==	930003xxxxxx9893

6.3 Attacking fixed padding RSA

6.3.1 Brute force attack using the public key

RSA is vulnerable to an attack called *chosen plaintext attack* in which the attacker can pick and choose any clear text and encrypt it using the public key. Since the public key is public it is assumed to be available to our attacker.

An attacker can mount a brute force attack by making a guess, G, on what the encrypted message might be, encrypt it using the public key RSA(G) and compare the result to the crypto text C, if they are equal the attacker has successfully recovered the encrypted message.

IF C*iphertext* == RSA(G*uess*) GOTO profit!

When using the RSA algorithm for encryption it is of monumental importance that the padding is random. When the padding is fixed or predictable the otherwise secure RSA algorithm is turned into a fancy one-way-function vulnerable to the same brute force techniques as any other one-way-functions. In fact the RSA algorithm is sometimes referred to as a *trap door one-way-function*.

6.3.2 Partial known plaintext attack

Truncated PANs

With access to a truncated PAN this attack is trivial. The one hundred thousand 2048 bit RSA computations needed to resolve the truncated IIN could be done in a few seconds on a modern computer.

Known IINs

Even without the truncated IIN a full-scale brute force on a specific IIN is still possible. Since there is no salt involved each encrypted f(G*uess*)can be compared to the entire list of C*ipherext*:s. As stated earlier it is easy to find lists of IIN:s on the Internet and thereby reduce the scope of the attack to roughly one billion (10^9) tries per IIN. It is not unlikely that an attack on a specific IIN could be reduced to 12 – 24 hours.

6.4 Possible solutions

6.4.1 PKCS#1

The PKCS standard #1 - *RSA Cryptography Standard* provides guidance for how to properly use and pad the RSA algorithm. Most of the available frameworks for using RSA, such as openssl, implement this standard.