The Design and Analysis of Cryptographic Application Programming Interfaces for Security Devices

Chapter 3: The Financial Cryptographic API

Version: 4.0

Issue Date: 2003-01-17

Table of Contents

List of Figures

List of Algorithms

3. The Financial Cryptographic API

3.1. Why Are We Interested?

The security of Automatic Teller Machine (ATM) and Electronic Funds Transfer at the Point of Sale (EFTPOS) networks was perhaps the most significant driver for modern cryptography, responsible for drawing it out of the secret government and military departments into the commercial world. It has been described as the so-called 'killer application' of cryptography [An01]. As a mature technology in a young field it presents a unique opportunity to study the phenomenon of the degradation of security over a prolonged time period.

Secondly, the concept of a 'PIN' is internationally understood and accepted. It competes with the use of passwords for the title of the most important identification technique. After all, everyone with a bank account and card has a PIN number and this was true before the advent of the Internet age.

The scale of use is impressive. It includes bank, credit and debit cards, card issuing banks and cards associations such as VISA, MasterCard, Europay and American Express. An enormous amount of money is protected daily by these security mechanisms.

3.2. Financial Security 101

3.2.1. Terminology

The account holder is the individual or entity that holds the account at the bank. We sometimes refer to the account holder as the user or customer. The bank, which issued the card and PIN to the account holder, is known as the issuing bank while the bank responsible for initially acquiring the transaction from the user (i.e., the account holder or customer) is known as the acquiring bank. PIN is an acronym for Personal Identification Number. This is a number that is used by the account holder to verify his identity to the issuing bank and is used as a means of authenticating a transaction. Associated with each user's account is an identifying account number known as the Personal Account Number (PAN). The bank uses the PAN to identify which account is being used in the transaction. PINs typically consist of between 4 and 12 decimal digits. Since knowledge of the PIN gives the right to transact with the account, it is imperative that the PIN be kept secret. This is done using encryption.

Prior to being encrypted, the PIN is formatted into an 8-byte buffer known as the clear PIN block (PB). The encrypted results is termed an encrypted PIN block (EPB).

3.2.2. Architecture

The diagram below (Figure 3-1) is a simple representation of an ATM or EFTPOS network. On the extreme left is the customer facing device, typically an ATM or point of sale (POS) device (although it could just as easily be a web browser, mobile phone or another of the new generation transacting interfaces). This device is connected to the bank that 'operates' the device, i.e., the acquiring bank. Note the user does not necessarily have to hold an account with the acquirer and for this illustrative example we assume that this is not the case. The acquiring bank must thus send the transaction to the bank with which the user does have the account (i.e., the issuing bank). The transaction is passed to an intermediary switch, which then 'switches' the transaction to the acquiring bank. Sometimes the switch does not have a direct link to the acquiring bank and so sends the transaction to second switch.

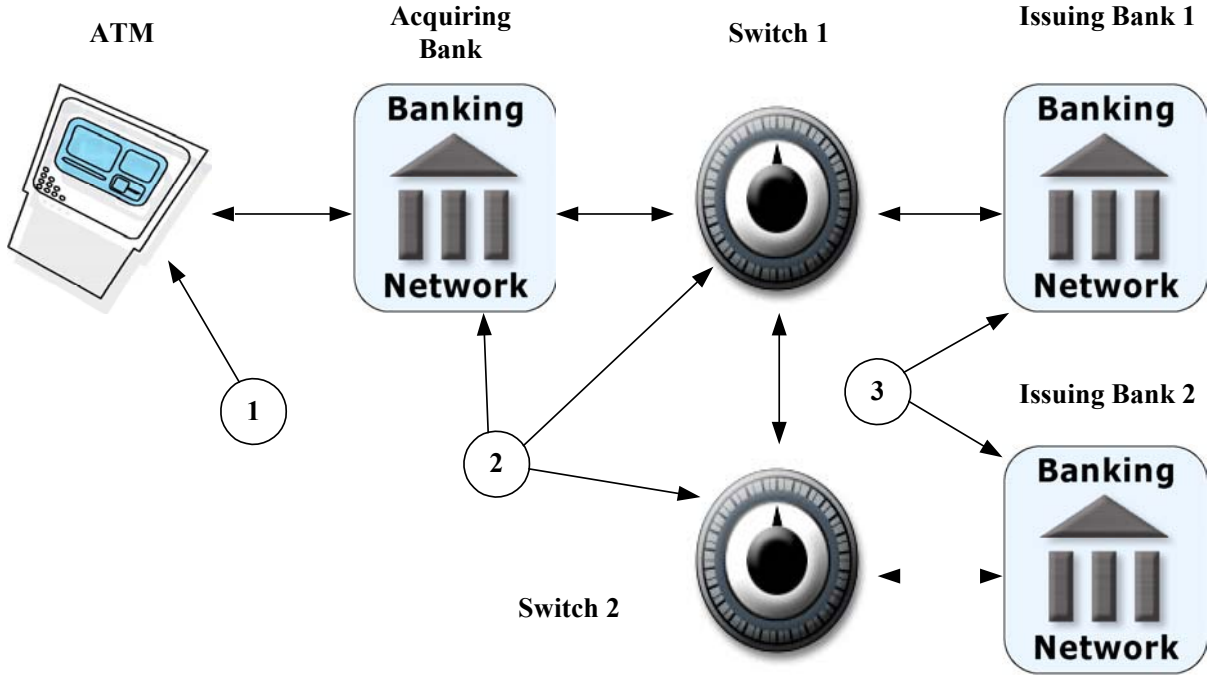

Figure 3-1: Simple representation of the EFT network

The reason for this architecture lies in the encryption methods used to protect the PIN. ATM and EFTPOS networks were first established in the late 70's and used symmetric key encryption only (typically DES). Thus any two entities that wanted to communicate with each other, first had to establish a shared key - normally done manually using key shares. The two entities are said to share a zone key. This process had three difficulties. Firstly a given bank could not transact with a (new) bank with which it had not yet established a shared key. Secondly, the process of establishing shared

keys with all the institutions with which it wanted to transact, was time consuming and costly. Finally, it required the secure storage of a large number of keys. Thus the use of switches arose to combat these problems. An acquiring bank needed only to share keys with one or more switches, which then provided the link from the issuing bank to the acquiring bank.

Let us map the path of the account holder's PIN through the system and identify the operations performed upon it. At point 1 on the diagram (the customer facing device) the user enters his or her PIN number. The PIN is immediately encrypted in the PIN entry device using the key shared with the acquiring bank and then sent to the acquiring bank. The bank must now forward the encrypted PIN to the switch. But the key shared between the bank and the switch is different, belonging to a different key zone from that shared by the acquiring bank and ATM. Thus the acquiring bank must first decrypt the PIN under the first key and then re-encrypt it under the key shared with the switch. The same process occurs at the switches and is marked by a 2 on the diagram. Finally, when the encrypted PIN arrives at the issuing bank, the bank must confirm that it is in fact the correct PIN associated with the account. Thus we have identified the three basic functions that must exist in our financial API, namely

1. PIN encryption,
2. PIN translation between zone keys and
3. PIN verification.

3.2.3. PIN Algorithms

3.2.3.1. Encryption

The PIN encryption is a trivial process. The PIN is formatted into an 8 byte clear PIN block and then encrypted using DES or 3DES (the block size of the cipher accounts for the 8 byte length of the clear PIN block). However, there exist a multitude of recognised formats for the clear PIN block.

3.2.4. PIN Block Formats

A fairly standardized notation is used to describe the possible formats whereby characters are used as placeholders for possible decimal and hexadecimal values. We reproduce the notation here for completeness.

X'y' denotes the hexadecimal character y and is equivalent to y_{16}. For example, X'0' = 0_{16} = 0000_2, X'1' = 1_{16} = 0001_2 and X'F' = F_{16} = 1111_2.

P = A 4-bit decimal digit that is one digit of the PIN value.
C = A 4-bit control value. The valid values are X'0' and X'1'.

```
L = A 4-bit hexadecimal digit that specifies the number of PIN digits.
F = A 4-bit field delimiter of value X'F'.
f = A 4-bit delimiter filler that is either P or F, depending on the
    length of the PIN.
D = A 4-bit decimal padding value. All pad digits in the PIN block
    have the same value.
X = A 4-bit hexadecimal padding value. All pad digits in the PIN block
    have the same value.
x = A 4-bit hexadecimal filler that is either P or X, depending on the
    length of the PIN.
R = A 4-bit hexadecimal random digit. The sequence of R digits can each
    take a different value.
r = A 4-bit random filler that is either P or R, depending on the
    length of the PIN.
Z = A 4-bit hexadecimal zero (X'0').
z = A 4-bit zero filler that is either P or Z, depending on the
    length of the PIN.
S = A 4-bit hexadecimal digit that constitutes one digit of a sequence
    number.
A = A 4-bit decimal digit that constitutes one digit of a user-specified
    constant.
```

One of the simpler formats is known as VISA Format 3. This format specifies that the PIN length can be a minimum of 4 digits and a maximum of 12 digits. The PIN start from the left most digit and ends by the delimiter ('F'). An example of a 6 digit PIN is:

```
VISA Format 3 PIN Block (PB) = PPPPPPFXXXXXXXXX
```

The most important (and complicated) format is ANSI X9.8 – the format specified by the ANSI standards on retail banking [ANSI X9.8]. This format is also known as ISO-0, VISA-1, VISA-4 and ECI-1. It is unique in that the 12 rightmost PAN digits are prepended with zeros and exclusive-ored (xored, $\oplus$) with the PIN to yield the clear PIN block

```
P1 = CLPPPPfffffffffFF
P2 = ZZZZAAAAAAAAAAAA
ANSI X9.8 PIN Block (PB) = P1 ⊕ P2
where C = X'0' and L = X'4' to X'C'
```

This was done for two reasons. It binds the account number to the PIN block and so provides a measure of protection against an incorrect account number being subsequently used. Secondly it diversifies the encrypted PIN block according to the account number. Even though two users may have the same PIN number, the associated encrypted PIN blocks will bear no resemblance since they have different account numbers. This prevents the so-called "code book" attack, in which an adversary makes use of a dictionary of encrypted PIN blocks.

The list of PIN formats includes:

- ISO-0 (ANSI X9.8, VISA-1, ECI1)

- ISO-1
- ISO-2
- VISA-2, VISA-3, VISA-4
- IBM 3624, IBM 3621, IBM 4700
- ECI-2, ECI-3
- Docutel

3.2.5. PIN Algorithms Continued

3.2.5.1. Translation and Reformatting

The translation function merely decrypts the PIN encrypted under the input key and then re-encrypts it under the output key before transmission to the target entity. What happens should this entity not support the PIN format used? The simple solution is an extension to the translate function called reformat which allows the user to specify a different format of the output PIN block. Should the output format be ANSI X9.8, the function needs to be able to specify a PAN as well.

3.2.5.2. Generation and Verification

PIN generation can be achieved in a variety of ways, either by means of an algorithm or chosen by a user (or both). An example of the algorithmic approach is the IBM 3624 PIN Generation Algorithm, which generates a PIN based on account- or person-related data, called the validation data. The validation data is enciphered under a PIN generating key, decimalised, and the desired number of digits selected as the PIN.

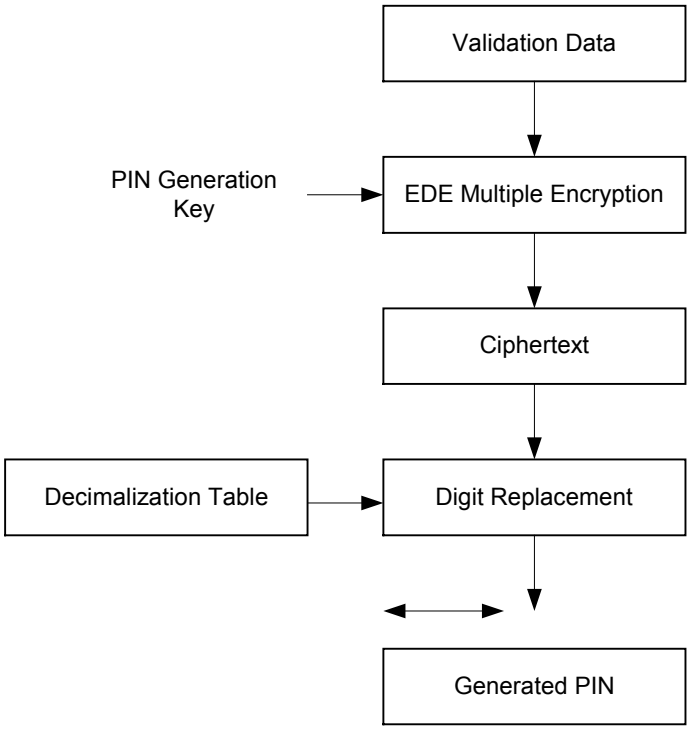

Figure 3-2: PIN generation algorithm

Verification is achieved by repeating the process (except that now the key may be called the PIN verification key) and comparing the calculated PIN with the PIN that is extracted from the encrypted PIN block.

3.2.5.2.1. Offsets

Generating algorithms can be extended to cater for chosen PINs. This is achieved through the use of offsets, which relate the algorithm generated PIN to the chosen PIN. The normal generation algorithm is run (with the same parameters) and the output called an intermediate PIN. The offset is calculated by subtracting modulo 10 some subset of the intermediate PIN digits (usually either the leftmost or rightmost digits) from the chosen PIN digits.

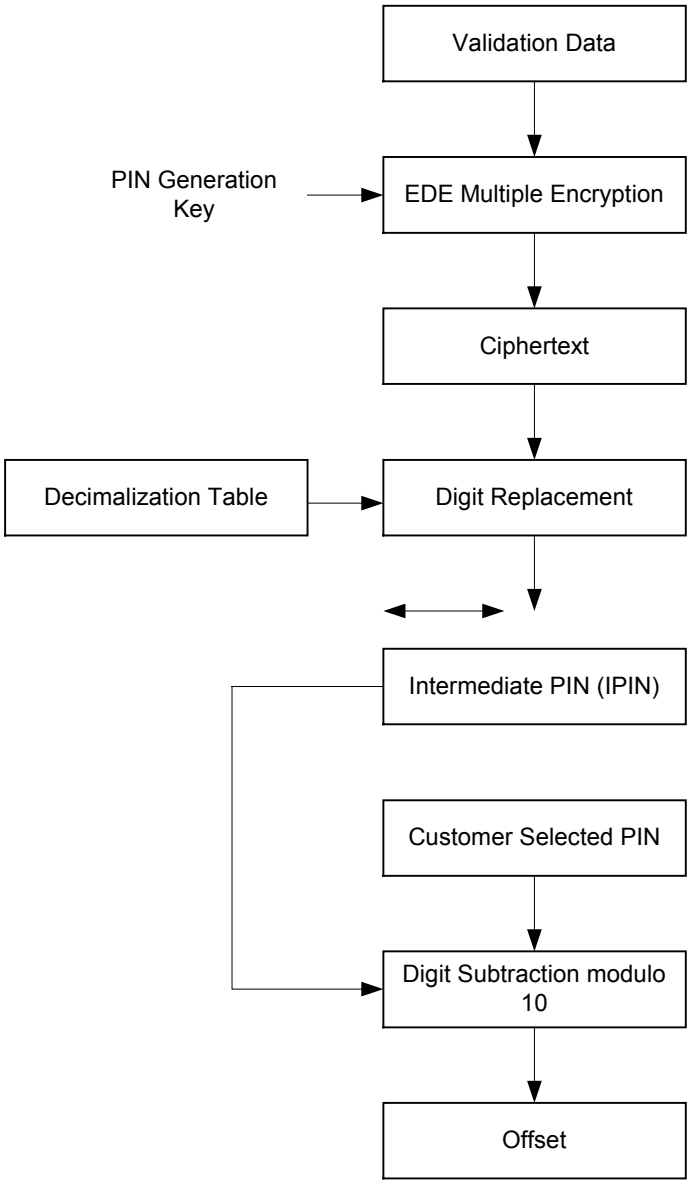

Figure 3-3: PIN offset generation algorithm

Verification requires that the offset be supplied to the call as well.

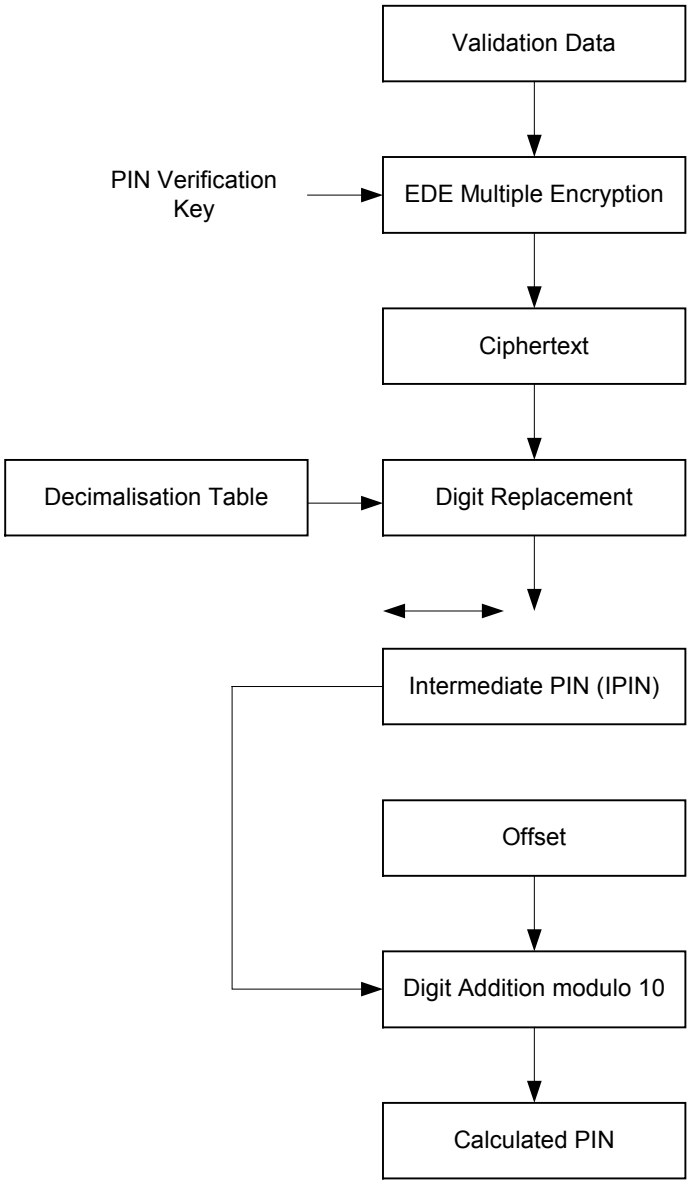

Figure 3-4: PIN verification algorithm

3.2.5.2.2. PIN Verification Values (PVV)

For a chosen PIN, the PVV generation algorithm accepts a PIN (either clear or encrypted) as input and must calculate a PIN verification value (PVV), which is stored. The algorithm is not unlike the PIN generation algorithm described in Figure 3-2 but with two significant differences. Firstly, the validation data has been replaced by the transformed security parameter (TSP). The VISA PVV algorithm encrypts the concatenation of 11 PAN digits, a key index (a digit in the range 1 to 6) and the first 4 digits of the PIN extracted from the encrypted PIN block (i.e. TSP = PAN || Key Index || PIN). Secondly, the result is decimalised in a unique way. Scanning from left to right, the first 4 decimal

digits encountered are returned as the PVV. Should there be less than 4 digits, a second scan is performed, in which the non decimal digits are converted to decimal digits (by subtraction modulo 10), and the first 4 resulting digits returned as the PVV.

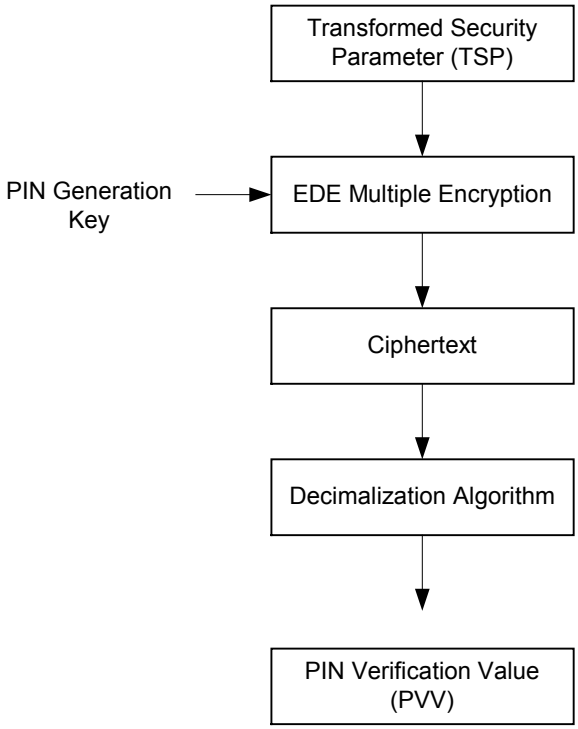

Figure 3-5: PVV generation algorithm

Verification involves extracting the PIN, calculating the verification value and comparing it to a supplied PVV. An example is the VISA PIN verification algorithm.

3.3. The Standard Financial API

3.3.1. Case Study of the CCA API

The PIN translate and reformat command in the CCA API (Release 2.40) is called CSNBPTR and has the following prototype. Note that the prototype has been slightly simplified.

```
CSNBPTR
{
  return_code,                    //Output – error code
  reason_code,                    //Output -  description of error
  input_PIN_encrypting_key_identifier,     //Input – old key
  output_PIN_encrypting_key_identifier,    //Input – new key
  input_PIN_profile,              //Input – the original PIN block format
```

```
  input_PAN_data,                //Input – the original PAN
  input_PIN_block,               //Input – the encrypted PIN block
  rule_array_count,              //Input
  rule_array,                    //Input
  output_PIN_profile,   //Input – the desired PIN block format
  output_PAN_data,      //Input – the desired PAN
  output_PIN_block      //Output
}
```

The `return_code` is a general error code with the `reason_code` indicating the reason for the error. The key identifiers (`input_PIN_encrypting_key_identifier` and `output_PIN_encrypting_key_identifier`) are the key tokens. The pin profiles specify the PIN block formats (e.g. ANSI X9.8, etc) and the `PAN_data` contains the PAN. The encrypted PIN blocks are called the `input_PIN_block` and `output_PIN_block` respectively.

The `rule array` is a list of parameters for the function (the `rule_array_count` indicates the number of parameters in the list). For this function, the caller can specify either `TRANSLAT` or `REFORMAT.` Translate indicates that only the PIN encrypting key is changed in the function, while the reformat option is used to change some combination of key, PIN block format or PAN.

The PIN verify command is known as CSNBPVR.

```
CSNBPVR
{
  return_code,                   //Output – error code
  reason_code,                   //Output – description of error
  PIN_encrypting_key_identifier, //Input – PIN encrypting key token
  PIN_verifying_key_identifier,  //Input – verification key token
  PIN_profile,                   //Input – PIN block format info
  PAN_data,                      //Input – the PAN
  encrypted_PIN_block            //Input – encrypted PIN block
  rule_array_count,              //Input – a count of the parameters
  rule_array,                    //Input – parameters for the call
  PIN_check_length,              //Input – length of PIN to check
  data_array                     //Input – the validation (or other) data
}
```

Here the `rule_array` indicates which algorithm to use to verify the PIN (e.g. IBM, IBM Offset, VISA PVV, etc). The `data_array` contains the validation data or transformed security parameter (depending on the algorithm).

3.3.2. Case Study of the Thales-Zaxus-Racal API

The Thales API supports a number of PIN encrypting keys types. These are

- *Terminal PIN keys (TPK)* which are used in the PIN encrypting terminal or device,

- *Zone PIN key (ZPK)* which are shared between entities sharing a key zone, and
- *Local Master keys (LMK)*.

The Pin verification key is known as the PVK.

There are a number of PIN translation functions that allow translation between each of the above key types. The basic prototype (ignoring key types) is as follows:

```
PIN Translate
{
  //inputs

  Source Key,               //source TPK/ZPK/LMK key
  Destination Key,          //destination TPK/ZPK/LMK key
  Maximum PIN Length,       //Value of 12
  Source PIN Block,         //the encrypted PIN block under the source key
  Source PIN Block Format,  //the format code for the source PIN block
  Destination PIN Block Format,  //the format code for the dest PIN block
  Account Number,           //the rightmost 12 digits of the PAN

  //outputs

  Error Code,               //e.g. 00 = no error; 20 = pin block data error
  PIN Length,               //the length of the returned PIN
  Destination PIN Block,    //the encrypted PIN block under the dest key
  Destination PIN Block Format
}
```

While the function does not support different source and destination account numbers (PAN), one can effect this operation by simply calling this function twice in sequence. On the first call we convert from the ANSI X9.8 format with associated source account number to a format without an account number. On the second call we translate back to ANSI X9.8 format, but specify the desired destination PAN.

The two PIN verification prototypes are also standard

```
PIN Verification Using Offsets
{
  //inputs

  PIN encrypting Key,       //TPK/ZPK/LMK
  PIN verification Key,     //PVK
  Maximum PIN Length,       //Value of 12
  Encrypted PIN Block,      //the encrypted PIN block
  PIN Block Format,         //the format code for the source PIN block
  Account Number,           //the rightmost 12 digits of the PAN
  Decimalization Table,     //the decimalization table
  Validation Data,          //the validation data
  Offset,                   //the offset

  //outputs
```

```
  Error Code                //e.g. 00 = no error; 01 = verification failed
}

PIN Verification Using PVV
{
  //inputs

  PIN encrypting Key,       //TPK/ZPK/LMK
  PIN verification Key,     //PVK
  Maximum PIN Length,       //Value of 12
  Encrypted PIN Block,      //the encrypted PIN block
  PIN Block Format,         //the format code for the source PIN block
  Account Number,           //the rightmost 12 digits of the PAN
  PVKI,                     //between 0 and 6
  PVV,                      //the pin verification value

  //outputs

  Error Code                //e.g. 00 = no error; 01 = verification failed
}
```

Hence we can conclude that the Thales PIN translate function matches our standard financial API functionality.

3.3.3. On the Validity and Applicability of the Standard Financial API

After comparing the functionality of the proposed standard financial API with the equivalent functionality found within the devices that dominate this market (including the CCA, Thales and Atalla APIs), we can conclude that our prototypes are indeed accurately reflective of the industry norms. We can thus proceed to the analysis stage with confidence knowing that the results will be applicable to the majority of relevant APIs.

3.4. Known Attacks and Assumed Level of Security

3.4.1. Exhaustive Key Search (Brute force)

Exhaustive key search is (almost) always a possibility. It provides an understood upper bound on the level of security that is hoped to be achieved - and so any technique that does not improve on it is of little interest. The strength of a secure algorithm is measured by the length of the key (effectively the size of the key space required to be searched). The current standard PIN transaction systems are 3DES [ANSI97, ANSI X9.8, ANSI X9.17, ANSI X9.24] - although many (typically historic) single DES systems still exist. Note that this is not an attack against correctness. In our current reality, a 3DES system is practically immune to such an attack.

3.4.2. Exhaustive Pin Search

The PIN space is considerable smaller than the key space. It is thus critical to prevent an adversary from being able to mount such an attack (since he would surely and rapidly succeed). It is probably for this reason that the ANSI X9.8 standard (Personal Identification Number (PIN) Management and Security) states "The system shall not be capable of being used or misused to determine a PIN by exhaustive trial and error". One obvious potential weakness would be any implementation, which accepts clear PINs to either the PIN verification or generation functions. An API that allows such functionality typically restricts it to a secure or authorized mode, thereby ensuring that it is not misused. At some point, the user has to be able to enter a PIN in the clear. However, this is a manual process (usually at a trusted interface) and hence cannot be easily automated. In addition, it is common practice to detect and prevent a user's trying of many combinations.

3.4.3. The Code Book Attack

To mount such an attack, the adversary will build up a 'code book' containing every PIN and the result of that PIN encrypted under a given key. To recover an unknown encrypted PIN, the attacker simply consults the codebook to find the encrypted PIN block and hence identify the associated PIN. Should the PIN be encrypted under a different key to the one used for the codebook, the attacker simply translates it to encryption under the codebook's key. As with the exhaustive key search, it should not be possible for the attacker to build up such a codebook and the same practical limitations apply.

For an ANSI X9.8 format n digit PIN, 10^n encrypted PIN blocks are required to be stored (for a given account number). This represents a trivial memory requirement and is insignificant to search. Naively, attacking formats containing random padding would appear to require a larger codebook, but this is not the case. All formats are 'equally' vulnerable, since the format can be changed in the translate (reformat) call (to the weakest one). The same technique can be used to 'eliminate' the variation offered by the PAN. It is noteworthy that regardless of format, key and pan, all encrypted pins are potentially vulnerable to a single codebook.

3.4.4. Key Separation Attacks

Key separation is a mechanism, which enforces that a given key is used as intended (described in Section 2.5.6). The well known attack scenario is supplying an encrypted PIN block and the PIN encrypting key to a standard data decrypt call, which would result in the clear PIN block being returned. This is an attack on the correctness of the transaction set and demonstrates the necessity to 'separate' PIN encrypting keys from data decrypting keys. There are two common methods for

achieving this. The first is to use different master keys for encrypting different types of keys. Incorrectly using an encrypted key would result in the key being decrypted under the incorrect master key, yielding a 'random' result. This may be detected if parity checking of keys is enforced or enabled. The second method, involves the creation of a variant of the master keys based on the type of key. Perhaps the most significant system is IBM's control vector method. A unique control vector is associated with each type of key. To encrypt or decrypt the key, the master key variant is created by exclusive-oring the master key with the control vector. Again, using an encrypted key as incorrect type, results in a 'random' result.

3.5. PIN Recovery Attacks

In this section we present a family of six new API attacks leading to PIN recovery. We describe these attacks in the context of the standard financial API. Thus we can apply them immediately to our reference APIs.

3.5.1. The Attack Model

We have the following inputs:

- *Query access* to a (potentially tamper proof) device with the typical PIN transaction set (our standard financial API as described above)
- An encrypted pin encrypting key which is valid for the device above (i.e., the PIN encrypting key is itself encrypted under a (master) key resident in the device)
- A valid encrypted PIN block (EPB), which was encrypted under the encrypted PIN encrypting key

It is our goal to find superior techniques to those listed under `Known Attacks', which exploit potential lack of correctness of the typical PIN transaction sets.

3.5.2. Our Results

We present 6 distinct attacks against the standard financial API functions that lead to the recovery of the PIN from an encrypted PIN block. These are

- the *ANSI X9.8 (ISO-0) attack* against functions to which the PAN is supplied,
- the *extended ANSI X9.8 attack* against translate and reformat functions,
- the *decimalization attack* against PIN verification algorithms using offsets,
- the *key separation attack #1* against PIN verification functions based on failure to enforce key separation between verification and translation (encryption),

- the *key separation attack #2* against PIN verification functions based on failure to enforce key separation for different verification algorithms, and
- the *check value attack* against PIN verification algorithms using the check value of a key.

The attacks are computationally trivial and extremely fast to implement with the capability of recovering many PINs per second, depending on the speed of the target device. Since the vulnerabilities identified by these attacks can be exploited in a number of similar variations, it can be viewed as families of attacks.

3.5.3. ANSI X9.8 (ISO-0) Attack

The significant input parameters for this attack are the

- Encrypted PIN Block (EPB)
- PAN
- Encrypted 'In' Key
- Encrypted 'Out' Key

Our attack strategy is as follows. In an iterative manner, we make a single modification to the PAN and observe the effects. Under normal operation, the encrypted PIN lock (EPB) will be decrypted and combined with the PAN to recover the PIN as per the following process:

INPUTS (EPB, P2)

1. $PB = d_k(EPB)$
2. $P1 = PB \oplus P2$
   $= 04PPPPFFFFFFFFFF$
3. Extract PIN as PPPP and test to confirm it is a valid PIN (i.e. each P is a valid decimal digit)
4. If test fails then exit with an error
5. Process the rest of the function as normal

Algorithm 1: Normal PIN extraction process

Since the PIN we are attacking is valid by definition, the function will complete normally. However, instead of supplying the correct PAN (effectively P2) to a call, we use a modified PAN ($P2' = P2 \oplus \Delta$) where say Δ = 0000x000000000. Now observe the same process:

INPUTS (EPB, P2')

1. $PB = d_k(EPB)$
2. $P1' = PB \oplus P2'$
   $= (P1 \oplus P2) \oplus (P2 \oplus \Delta)$
   $= P1 \oplus \Delta$
   $= 04PPPPFFFFFFFFFF \oplus 0000x00000000000$
3. Extract PIN as PPPP $\oplus$ 00x0 and test to confirm it is a valid PIN

```
4. If test fails then exit with an error
5. Process the rest of the function as normal
```

Algorithm 2: Basic ANSI X9.8 attack method

So if (P ⊕ x) is a decimal digit then the call will proceed normally and pass. If (P ⊕ x) is not a decimal digit then we expect the call to fail and return an error code. Thus we have a test for (P ⊕ x) < 10.

Our next step is to turn this test into an algorithm to identify P. The simple approach is to try all possible values of x, yielding a semi-unique pattern of 'passes' and 'fails', allowing us to identify P. We observe that for $y \in Z_{16}$, $y < 10$ if and only if $y \oplus 1 < 10$. Thus, for any given number $x \in Z_{16}$, both x and $x \oplus 1$ will result in an identical pattern of 'passes' and 'fails' and, furthermore, no other values of x will yield the same pattern. A more sophisticated approach is to build a decision tree.

This attack will work against any function that accepts the encrypted PIN block as well as the PAN and (attempts to) extracts the PIN. Thus it can be applied against both the verification, translation and reformatting functions.

3.5.3.1. Work Factor

The simplest (and least efficient) algorithm requires 16 calls per PIN digit for a total of $\sum_{i=1}^{n-2} 16 = 16 \cdot (n-2)$ to recover the last $n-2$ PIN digits.

3.5.4. Extended ANSI X9.8 Attack

The previous attack had two limitations, namely:

1. it identified each PIN digit as P as belonging to the set $\{P, P \oplus 1\}$, and
2. did not recover the first 2 PIN digits.

We now consider methods to extend this attack.

The attack mechanism involved manipulating (i.e. lying) about the value of the PAN associated with the encrypted PIN block. The logical extension is to ask the question whether there are other parameters to these functions that can be manipulated? The answer is yes - the PIN block format. Consider the effect of submitting an ANSI X9.8 formatted PIN to the reformat function *but specifying*

it as a PIN block of another format (e.g. a VISA-3 format). To simplify the analysis, we assume a NULL PAN (i.e. PAN = 000000000000).

INPUTS (EPB, P2, Format')

1. The clear PIN block is obtained by decrypting the encrypted PIN block (i.e. PB = d_k(EPB)). Since the clear PIN block is actually an ANSI X9.8 format, hence
   PB = P1 ⊕ P2
   = 0LPPPPFFFFFFFFFF ⊕ 0000000000000000
   = 0LPPPPFFFFFFFFFF
2. The PIN is extracted according to the VISA-3 formatting rules (i.e. PIN is extracted as 0LPPPP.
3. The PIN is then formatted into a new PIN block as
   P1 = 0L'0LPPPPFFFFFFFF (where L' = L+2)
   P2 = 00000000000000000
   PB = P1 ⊕ P2
   = 0L'0LPPPPFFFFFFFF (where L' = L+2)
4. The encrypted output is EPB = e_k(PB))

Algorithm 3: Increasing PIN length

In this process we have extended the original PIN to a new value 0LPPPP. By doing this, we have now exposed the first two PIN digits to the previous ANSI X9.8 attack.

Consider now the effect of using a non-null PAN (Δ) at the same time. The clear PIN block is obtained by decrypting the encrypted PIN block (i.e. PB = d_k(EPB)). Since the clear PIN block is actually an ANSI X9.8 format,

PB = P1 ⊕ P2
= P1 ⊕ Δ (say Δ = 00000x0000000000)
= 0LPPPPFFFFFFFFFF ⊕ 00000x000000000000

The PIN is extracted according to the VISA-3 formatting rules, which means we expect the hexadecimal digit 'F' to mark the end of the PIN. There are 3 possible outcomes:

If P ⊕ x < 10 (i.e. a decimal digit) then the PIN will be extracted as the six digit PIN 0LPPPP ⊕ 00000x. The call will proceed normally.

If (P ⊕ x) = F then the PIN will be extracted as the 5 digit PIN 0LPPP. The call will proceed normally.

However 10 ≤ (P ⊕ x) < 'F' is interpreted as a format error resulting in the call failing.

This method allows us to identify uniquely the value of the PIN digits.

3.5.4.1. Work Factor

Again, the simplest (and least efficient) algorithm requires 16 calls per PIN digit for a total of $\sum_{i=1}^{n} 16 = 16 \cdot n$ queries plus the initial call to expose the first two PIN digits. A typical 4 digit PIN is thus recovered in $16 \cdot 4 = 64$ queries.

3.5.5. Decimalization Attack

The significant input parameters to the decimalization attack are the

- Encrypted PIN Block (EPB)
- Validation Data
- Decimalization Table
- Offset
- Encrypted PIN Encrypting Key
- Encrypted PIN Verification Key

Our attack strategy is as follows. In an iterative manner, we make a single change to an entry in the decimalization table and observe the effects on the offset. We first demonstrate this by means of an example.

Example.
We start off with a PIN (6598), a PIN verification key, some validation data, a decimalization table and the resulting offset (2117).

```
PIN           = 6598
PIN Ver Key   = 05050505 05050505
Val. Data     = 11223344 55667788
Ciphertext    = E481FC56 58391418
Dec. Table    = 01234567 89012345
IPIN          = 4481
Offset        = 2117
```

We now make a single change to the decimalization table replacing the first entry (which previously mapped a 0 in the ciphertext to a 0) with a new value of 1 (i.e. now maps a 0 in the ciphertext to a 1).

```
Dec. Table (0)= 11234567 89012345
```

Since the ciphertext does not contain a 0 in the first 4 nibbles, there is no effect on the value of IPIN and so the original offset will continue to pass.

```
IPIN          = 4481
```

```
Offset          = 2117 (will pass)
```

Next we replace the second entry in the decimalization table with a 2 (i.e. it now maps a 1 in the ciphertext to a 2).

```
Dec. Table (1)= 02234567 89012345
```

Since the ciphertext does contain a 1 in the 4th nibble, this results in a change to the IPIN value.

```
IPIN            = 4482
```

This time the original offset of 2117 value will fail. However the value 2116 will pass.

```
Offset          = 2117 (will fail)
                = 2116 (will pass)
```

Thus we have identified that the 4th digit in the original IPIN is a 1 and hence that the 4th PIN digit is 1+7 = 8 (IPIN + Offset).

This process can be explained more formally as follows. Since the user supplies the decimalization table to the call, one can repeatedly query the target with modifications to the table. Suppose we know the offset for a given PIN block (using a given decimalization table). Consider the effect of changing a single element in the table (the k^{th} entry in the table mapping $\{i \rightarrow j_i, i \in Z_{16}, j_i \in Z_{10}\}$ to a new value j_k')

If the hexadecimal digit k was not found in the first n digits of ciphertext, then there is no change in the value of IPIN, and the same value of the offset will pass the verify call. However, for each instance of k in the ciphertext, the corresponding digit of IPIN will be remapped to j_k'. The original offset will now fail the verify call. Using this approach we can identify the possible values of the hexadecimal digits in the first n digits of the ciphertext.

This technique can be further strengthen by setting $j_k' = j_k + m$ where m is a known (non zero) value (addition is modulo 10). Due to the simple relationship between the offset and the IPIN, we know that by adding m to a digit in the IPIN, the corresponding digit in the offset is reduced by m. Thus we can search through all possible ciphertext digit locations that contain k, by modifying the offset value and supplying the modified offset and decimalization table to the verify function. After at most 2^n queries, we will have identified all digit locations in the ciphertext with the value i. By repeating through all possible values of k, we can uniquely determine the value of the first n digits of ciphertext and thus IPIN. Again since we know the offset and IPIN, we can trivially calculate the value of the PIN.

INPUTS: DEC. TABLE

1. Search for offset (using the decimalization table i → i mod 10)
2. For k = 0..15
   2.1. Replace the entry k → k mod 10 in the original decimalization table with k → k + m mod 10
   2.2. For each possible location of k in the ciphertext (including none).
      2.2.1. Test the corresponding modified offset.
      2.2.2. If 'pass', then store locations of k in ciphertext.
3. Decimalise the 'recovered' ciphertext and store as IPIN
4. Return PIN = IPIN + OFFSET

Algorithm 4: Decimalization Attack

3.5.5.1. Work Factor

The initial search for (an unknown) n digit offset requires at most 10^n queries (although it may be possible to reduce this to $10^4 + (n-4)\bullet 10$ queries using the method described in Section 3.5.5.2). Each change to the decimalization table requires at most 2^n queries (again it may be possible to reduce this to $2^4 + (n-4)$ queries). At most we need to try 15 of the 16 entries in the table for a maximum total of $15\bullet 2^n$ queries. The actual attack time is dependant on the speed of harvesting the data. An attack against a 4 digit PIN using a known initial offset will require at most 240 queries or less than 0.25 seconds for a device capable of 1000 transactions per second (TPS). A low-end device (10 TPS) will take at most 24 seconds.

3.5.5.2. Improving the Search for Offsets

Some APIs allow the caller to specify the length of the offset (i.e., the length of the offset is treated as independent of the length of the PIN). The only restriction is that the offset is at least as long as the minimum possible length of the PIN, traditionally 4 digits, and does not exceed the length of the PIN. In this case, an attacker can initially search for a 4-digit offset (requiring 10^4 queries). Thereafter, each subsequent digit can be searched independently by specifying incrementally increasing offset lengths. This search method requires a total of $10^4 + (n-4)\bullet 10$ queries to recover a n -digit offset.

3.5.6. Key Separation Attack #1 (Failure to separate PIN encryption and verification keys)

We describe an exhaustive PIN search attack that exploits a lack of separation between PINGEN/PINVER and IPINENC/OPINENC keys to perform the search. Since we have no separation, we may interchange the use of the pin encrypting key and the pin verification key. We

wish to recover the PIN (P), encrypted under some key k. We have the associated encrypted PIN block EPB and the encrypted key k.

Our first step, is to translate the EPB to ANSIX9.8 with PAN = 000000088888. The clear PIN block is of the form PB = $04P_1P_2P_3P_4FFFFF77777$. Let $P^i = P^i_1P^i_2P^i_3P^i_4$ be a guess for P (for simplicity of notation, we have used a 4 digit pin). We format a 16 hexadecimal validation data string as VAL = $04 P^i_1P^i_2P^i_3P^i_4FFFFF77777$. We now use this validation data as input to say the IBM Pin Generation algorithm using the encrypted key k. The validation string is encrypted under the clear value of k, decimalised and the first 4 digits returned as the generated PIN (P_{gen}). We simply compare the first 4 decimalised digits of our translated encrypted PIN block to P_{gen}. If $P^i = P$, then the values will be equal. We can expect multiple collisions due to the decimalization process and the fact that we only have 4 digits to compare. However, we can eliminate false witnesses by repeating the process and modifying the validation data and the PAN (e.g. use PAN = 000000088887 and VAL = $04P_1P_2P_3P_4FFFFF77778$). Simply by iterating through all possible values of P^i we can identify P. Alternatively, we can build up a codebook.

One may question why we didn't use a null PAN, and format the validation data as VAL = $04P_1P_2P_3P_4FFFFFFFFF$. The reason lies in the fact that some API's take as input, effectively an 11 hexadecimal digit validation string, and extract the last 5 digits from the rightmost 5 digits of the PAN. Other API's require the 12^{th} digit to specify a key index in the range 1 to 6. The description above will satisfy the addition requirement that the PANs supplied to generate and translate functions consists of decimal digits (for the above description, the generate call uses a PAN = 000000077777).

A limitation of this attack, is that it requires access to the generate function. However, as mentioned before, this is a security risk in itself, usually protected against by restricted its usage to a secure or authorized mode, or alternatively, encrypting the output (or requiring encrypted inputs). If the output is encrypted, then we must obtain the first 4 decimalised digits of the encrypted PIN block as an encrypted PIN for comparison. This in itself is perhaps not a strenuous requirement. However, there is a more powerful version of the attack, using the verify function (which is unlikely to have access control restriction). We proceed as before, generating the validation as before, but supplying it to the verification function (using offsets) and supplying the encrypted pin block. We require one extra piece of information, namely the offset. If $P^i = P$, the validation data is identical to the clear pin block as before, the intermediate PIN will equal the first 4 decimalised digits of the encrypted PIN block (which is known). We denote this value by IPIN.

The offset is the difference of the intermediate PIN and the clear PIN modulo 10. Hence the offset will equal will equal the difference of P^i and IPIN modulo 10. Thus for each value P^i we calculate OFFSET = P^i - IPIN and supply it to the verification call. If P^i = P, the call will pass. False witnesses can be eliminated as before.

3.5.6.1. Work Factor

Essentially a method to search the PIN space, this attack therefore requires 10^n queries to identify an n digit PIN given knowledge of a suitable offset. Searching for the offset requires 10^n queries (although it may be possible to reduce this to 10^4 + (n-4)•10 queries using the method described in Section 3.5.5.2).

3.5.7. Key Separation Attack #2 (Competing Verification Algorithms)

In this attack, we play two different verification algorithms against each other. This is a fascinating result since it shows that taking two (potentially) individually secure verification functions, and allowing both in a single API can destroy its security. It serves as a warning for designers (and implementers) of API's against blindly adding functionality, even though the function may be secure on its own. It also shows the need for extremely fine granularity of separation of keys associated with different functions, even though the functions are of a similar nature.

The VISA PVV algorithm encrypts the concatenation of the 11 digit transformed security parameter (TSP), a key index (a digit in the range 1 to 6) and the first 4 digits of the PIN extracted from the encrypted PIN block (i.e. TSP || Key Index || PIN). The result is decimalised in a unique way. Scanning from left to right, the first 4 decimal digits encountered are returned as the PVV. Should there be less than 4 digits, a second scan is performed, in which the non decimal digits are converted to decimal digits (by subtraction modulo 10), and the first 4 resulting digits returned as the PVV.

The probability that the first 4 digits are indeed all decimal digits (and hence form the PVV) is $(\left(\frac{10}{16}\right)^4 = 0.153)$. Thus for a given key, by trying 7 different values for the TSP, we can expect one to exhibit this property.

Consider now the result of supplying the value (TSP || Key Index || PIN) as validation data to the (IBM) offset algorithm under the same verification key and using a decimalization table that maps n $\rightarrow$ n mod 10. The first 4 digits recovered from the encryption are the same as the PVV, and

unchanged by the IBM decimalization process (i.e. IPIN = PVV). These 4 digits are subtracted from the clear value of the PIN to obtain the offset (or OFFSET = PIN - PVV).

It is possible to use this relationship to search for the PIN, by iterating through all possible value for the PIN (denoted P^i), and testing whether $OFFSET^i = P^i$ - PVV, satisfies the verification algorithm with validation data (TSP || Key Index || P^i). As indicated earlier, the entire process needs to be repeated 7 times on average to witness the true value for the PIN. We can expect some false witnesses for the PIN but these can be eliminated by increasing the number of repetitions (say to 14 when we can expect the real PIN to have been witnessed twice).

INPUTS: Encrypted PIN Block

1. Loop i: (i = 1..7)
   1.1. TSP^i <= i || 1 || 1234
   1.2. Search for PVV^i
   1.3. Loop j: (j = 0...9999)
      1.3.1. PIN^j = j
      1.3.2. $OFFSET^{i,j} = PIN_j - PVV^i$
      1.3.3. VAL $DATA^{i,j}$ = i || 1 || j
      1.3.4. Test if $OFFSET^{i,j}$ verifies the PIN, if so then j is a candidate.

Algorithm 5: Key Separation Attack against competing verification algorithms

Example:

```
PIN          1234
Verification Key (K) 0505050505050505

i = 1:
TSP[1]           10000000000011234
E(K)(TSP[1])     D84355EF089EF293
PVV[1]           8435
Will not correctly witness the PIN.

i = 2:
TSP[2]           2000000000011234
E(K)(TSP[2])     9D15A1C0BE1DD129
PVV[2]           9151
Will not correctly witness the PIN.

i = 3:
PVV[2]           3000000000011234
E(K)(TSP[3])     6787FA69080E3C93
PVV[3]           6787
Will correctly witness the PIN.
```

```
  j = 1
  VAL_DATA[3,1]   = 3000000000110001
  OFFSET[3,1]     = 4324
  PIN_verification call returned FALSE

  ...

  j = 1234
  VAL_DATA[3,1]   = 300000000011234
  OFFSET[3,1]     = 5557
  PIN verification call returned TRUE.  Store j = 1234 as candidate for the
PIN.

  ...
```

Changing the decimalization table and calling the offset verify function again can eliminate some false witnesses (the digits 0 to 9 must remain mapped to themselves). For example, let $n \rightarrow n$ ($n < 10$), else $n \rightarrow n+1 \bmod 10$ ($n \geq 10$).

3.5.7.1. Work Factor

The inner loop of the algorithm requires 10000 calls to the offset verify function plus the initial search for the PVV, which requires a further 5000 calls on average (10000 maximum). The outer loop needs to run 7 times on average to witness the correct value of the PIN. In practice, we run this loop 14 times to eliminate false witnesses for a maximum possible total of $14 \cdot (10000 + 10000) = 280000$ queries.

3.5.8. Check Value Attack

The check value function encrypts a 64 bit binary zero under the supplied key. The PIN verify function (for IBM and GBP algorithms) encrypts a 64 bit user supplied validation data. The resulting ciphertext is decimalised via a user supplied decimalization table and termed the intermediate PIN (IPIN). The offset is the result of subtracting the IPIN from the customer selected PIN modulo 10 (OFFSET = PIN - IPIN modulo 10). The key observation is the similarity in operation between the check value function and the verification function.

Consider the case when the validation data is a 64 bit binary zero. The result of the encryption stage is the same as the result from the check value call (or more accurately, the first n bytes are the same for a n byte check value). The decimalization table is known, so it is trivial to calculate the IPIN value. It is also possible to recover the value of the offset by exhaustive search (again using the verify function). With knowledge of both the IPIN and offset, it is a trivial calculation to determine the PIN.

Algorithm:

1. Calculate the check value of key
2. Decimalise the check value and store as IPIN
3. Search for the OFFSET (with validation data = 000000000000000)
4. Return PIN = IPIN + OFFSET

Example

```
PIN                  1234
PIN ver key          0505050505050505

Validation data      0000000000000000
Decimalization Table 0123456789012345
OFFSET(searched)     3034

Check Value          8CA0964
IPIN                 8200 (= the check value decimalised)

Calculated PIN       1234 (= OFFSET + IPIN)
```

Some observations: The attack (as always) accepts an encrypted PIN block and its associated (encrypted) PIN encrypting key. The actual value of the PIN verification key is not important and hence one can merely conjure the key (if possible) or use any other PIN verification key in the system.

3.5.8.1. Work Factor

The attack is fairly efficient, requiring only a single call to the key test function (to obtain the check value) and a single search for the offset. For an n digit PIN this requires at most 10^n queries (although it may be possible to reduce this to $10^4 + (n-4) \bullet 10$ queries using the method described in Section 3.5.5.2).

3.6. Analysis of the Commercial APIs

3.7. Real World Implications.

The question that begs asking is whether these attacks are a credible real world threat. The first comment that should be made (and typically the first response from a vendor susceptible to a newly published vulnerability [IBM01]) is that real world systems should be following standard industry best practices that, if implemented correctly and enforced, should limit a potential hacker's ability to perform such attacks. This is typically taken to mean the use of physical access control and operation

procedures to restrict access to the security devices. This defence is correct in that an attacker does require some form of query access to the device (whether it be physical or remote electronic access) in order to effect the attack.

However there are some observations and counter arguments that question the validity of this defence. The first is that an attacker can attack at weakest point. By the nature of the financial systems' architecture, this means that one institution's account holder can be compromised on another institution's network (i.e., as the encrypted PIN travels through the network belonging to a different institution, it may be attacked). Hence one must guarantee that all potential networks through which the PIN may travel are secure. Can one realistically expect, claim or prove that all possible institutions in all cases correctly implemented and observed the required best practices? And how would one justify buying an expensive tamper ***resistant*** security module in the first place if one's defence rests on physical access control?

In addition, these devices now find use in other environments such as multi-lane retail stores where they are used as concentrators for the EFTPOS devices in the store. While it may be reasonable to expect a financial institution to have the necessary rigorous controls in place, it may not be necessarily be the case in a high volume, public store.

Finally, there is an implied assumption as to the impenetrability of the network in discounting the possibility of a remote attack. A traditional hacker gaining access to host with a security device could easily implement the attack.

3.8. Solutions

So what went wrong? Firstly, it is clear that a number of functions were just badly thought out and are insecure. The ease with which decimalization tables can be manipulated is perhaps a good example. Secondly, individually secure functions were added to the API in a manner to make the entire system insecure. The use of check digits function to compromise the verification algorithm is a classic example. Insufficient attention was given to the possible interplay between functions such as the key separation with competing verification algorithms. Finally, the absence of a single standard, to which everyone completely adheres, contributed to the complexity of the system, creating opportunities to exploit (e.g. the many different formats and algorithms that exist due to historical reasons). Saying this, it is difficult for a vendor to restrict functionality when different customers want different functionality from the same product.

Our proposed solutions are the following.

1. **Remove 'weaker' algorithms and functions and adhere to a single standard**

   When there is a choice between algorithms that effectively provide the same functionality, the weaker one should be removed. By leaving only the strongest function the level of security has immediately been increased. This approach complements the goal of supporting only a single standard to the exclusion of everything else. It reduces the possibility to interaction between algorithms.

2. **Parameter (data) Integrity**

   Many of the attacks exploit the possibility of modifying input parameters to the functions (including, amongst others, the PAN, the PIN block format, decimalization tables, validation data, transformed security parameter (TSP), etc). Encrypting the parameters offers a possible solution. However perhaps the most logical solution is to include the use of MACs (or similar methods) to ensure data integrity. For example, a decimalization table is supplied to a function call along with a MAC and the key for verifying that MAC. It may be natural to expand the PIN block, adding fields to include the PAN and PIN block format and a MAC over the entire PIN block (or encrypted hash). A clever solution to enforcing the correct PIN block format is the use of an additional control vector that is unique per PIN block format. This is exclusive-ored to the key before use. This method known as PIN block variance control was found in an earlier IBM API known as ICSF. The advent of AES may provide an option for extending the PIN block owing to its increase block size (128 bits instead of 64). As a result, a PIN block of the form Length||PIN||PAD||PAN||Format could be used, eliminating many of the problems.

   These solutions are fairly obvious. The challenge comes in ensuring interoperability between devices from different manufacturers.

3. **Key Separation**

   Key separation is of vital importance. As mentioned before, it is a well-understood principle. Failure is typically either as a result of a design or implementation oversight or alternatively as a lack of understanding of the extent of separation required.

While implementation oversights are potentially inevitable (and not limited to key separation issues) they can be addressed through training and peer review. A structured and complete approach is required to identify possible areas of compromise in design where there exists the possibility of compromising interactions.

Similar functionality (such as slightly different PIN verification algorithms) is often naively grouped together, for example as parameterised functions or supporting the same key types. While this may seem intuitively acceptable, in our analysis we have shown this be a source of separation issues. Separation must be extremely fine grained down to the individual algorithm level and the design must resist the temptation to categorize on a more general level. In the case of PIN verification, this means that keys must be identified and restricted for use with only a single verification algorithm as opposed to the current common practice of being useable with a number of algorithms. Likewise a distinction between PIN encrypting and PIN verification keys must be enforced.

The use of PIN block variance (essentially a key separation mechanism that relates PIN block format to the key) is a useful method offering the fine-grained restrictions required of an API.

4. Electronic access control

The benefit of electronic access control cannot be overstated. It should be fine grained, allowing the individual enablement/disablement of:

- formats,
- algorithms, and
- functions.

Whenever working with access control one must observe the mantra of permitting only the barest minimum required functionality. Start by disabling everything. Then enable only what is required.

Access control has the exceptionally useful property of being able to disable (or remove) a function that has been shown to be insecure. As such it provides an immediate defensive response by removing vulnerability in the device. It can also be effective in enforcing dual control.

5. Implementation

As has already been alluded to, a number of issues can arise as a result on implementation errors and oversights. Building a system that is expected to withstand a high level of malicious or 'out of specification' operation by malicious users, is a complex and challenging task and famously referred to as 'Programming Satan's computer' [AN95]. An engineering or development team working on such an assignment requires an emphasis on security priorities and quality control which at the very minimum requires specialized training and a submission to peer review.

3.9. Hackers and Threats: Real World Scenarios

We diverge briefly from the theoretical analysis of API vulnerabilities to explore the exploitation thereof and the impact of such failures in the real world. Thus we present a series of scenarios in which we leverage our knowledge to attack financial institutions and their account holders.

3.9.1. Insider Attack

The insider attack is the first and obvious attack arising from the weaknesses in the standard financial API. It presupposes the existence of an 'insider' who is either a person internal to a given financial institution (e.g. employee, contractor, cleaning staff, etc) or else an individual who has gained access to the financial network, perhaps through some traditional network hacking technique. This attacker either has knowledge of the attack methods described or else has downloaded exploit code available on the Internet. He begins by monitoring the transaction stream for a period, recording the encrypted PIN blocks and associated account numbers that pass through the system. Using the exploit code, the clear PIN numbers are extracted through a sequence of queries against the hardware security module. Having obtained a list of account numbers (and other information) along with the associated PIN numbers, the attacker surreptitiously leaves the scene of the attack.

To exploit this list, the attacker purchases a set of 'white cards' (blank cards without any silkscreen logo) and a card reader/writer (available as a combined package on the internet for $595 [HH]). He then writes the account information for each harvested account onto a card, creating in effect a duplicate card of the original card still in the customer's possession. He distributes these cards to a network of accomplices (perhaps a gang of street kids) and encourages them to perform a random tour of ATMs in the area. The instruction is that each card be used once a day, at a random ATM and used to withdraw the estimated daily withdrawal limit.

Let n be the number of compromised accounts, p the average period before unnatural transaction behaviour is noticed and l the daily withdrawal limit. The total fraud value is $F = n \cdot p \cdot l$.

Example:

$n = 5000$

$p = 2$

$l = \$1000$

Total Fraud = $ 10,000,000

Harvesting a large number of accounts (like the 5000 in our example) is trivial due to the efficiency of the attacks. A Trojan or piece of software could harvest many millions of accounts over just a few days – limited only by the transaction throughput. Log files could potentially provide an immediate and large source of encrypted PIN blocks.

There exist other possible methods to extract monies from the compromised accounts such as the use of Internet banking or account transfers. However, with the exception of bill numbers on the currency, there is limited ability to trace the money after the attack. Regardless of the method employed, a wise adversary would immediately launder the money.

3.9.2. Account Holder Attack

While the insider attack was the work of a third party that defrauded an account holder, this attack provides a mechanism for the account holder to defraud the institution. The malicious account holder first produces a number of duplicate 'white cards' of his own card. These he distributes to multiple accomplices, preferably in different geographical locations and instructs them to perform a random tour of the ATMs in the region. Upon receiving the first bank statement with these transactions on them, he immediately reports the 'unauthorized' activity on his account and disputes the transactions.

A first point is that it may be advisable for the malicious account holder to perform a valid transaction "simultaneously" with the fraudulent ones since this 'proves' that the account holder is in possession of his card at the same time as the fraudulent transaction occurred and preferably was in a different physical location. This provides fairly significant circumstantial evidence that the account holder was not involved in the fraud.

Extending this further, it is best done by multiple cardholders from a given institution since

1. it shows that it was not an isolated incident,
2. hence raises questions the security of the institution and

3. gives the impression of a possible insider attack.

Again let n be the number of compromised accounts, p the average period before unnatural transaction behaviour is noticed and l the daily withdrawal limit. The total fraud value is $F = n \cdot p \cdot l$ and the average return per individual is $R = p \cdot l$

Example:
$n = 100$
$p = 10$
$l = \$1000$
Total Fraud = $ 1,000,000
Average return = $ 10,000

3.9.3. The Repudiation Attack

The repudiation attack is an extension of the account holder attack. Here the malicious account holder simple denies a transaction. This will result in the bank's normal dispute resolution procedure being followed leading to possible litigation. The attacker will disown the transaction and claim that it must be an error on the bank's part or deliberate fraud and argue the insecurity of the system. The approach is best if the security of the institution has already been questioned.

For example, following a successful account holder or insider attack being made public – other account holders (acting individually) may dispute valid transactions that occurred during (or after) the attack period. The financial risk is great due to the possible scale (e.g. 0.1 % of an institution's 1,000,000 customers each disputing a $1000 transaction would result in a combined value of $1,000,000).

Perhaps more significant would be the loss of confidence in the given institution which could well be more damaging than any dollar loss. This leads us to some other approaches to benefit from these attacks.

3.9.4. The Competitor Attack

As mentioned earlier, an account holder is able to use his or her card on networks other than the account issuing bank's network. As a result, an account holder can be attacked on any network through which his PIN travels. This includes competitors' networks.

Thus a competitor could choose to mount an attack against account holders of a particular institution. In addition, as administrator of their own network, is it possible for them to use the additional benefits and powers that are associated with administrator privileges to effect this. Combined with unlimited time and access, the success of the attack is guaranteed.

However, the reward is not the stolen money but rather the 'after effects'. The negative publicity and damage to customer relationships that would arise after such an attack could destroy a bank's credibility and customer base. This resulting fall out would benefit the competitor. This has the advantage that there is less of a connection between accomplices and the attacker (in this case the competitor institution). There is no incriminating cash trail leading back to the attacker and no need for laundering. In addition, the effective monetary value of the reward could be considerable.

3.9.5. The Stock Market Attack

In another variation that eliminates the cash trail and potentially increases the value of the reward, the attacker sells the stock 'short' prior to any attack (a stock market strategy that pays off should the stock price fall). The negative after effects could contribute to a dramatic decrease in stock price.

3.9.6. The Anarchist Attack

With the goal of creating confusion and disorder in the world's financial institutions, an anarchist would employ all and any combinations of all the previous attacks.

3.10. Looking Ahead

Question: How can a bank guard against such attacks?
Answer: The bank should:

1. Contact its vendor, request any best practices information and implement it.
2. Be vigilant. Increase its auditing.
3. Reassure its clients.

4. Wait.
5. Apply positive pressure on the role players.

Question :Is that all? Why can the bank not just solve the problem?
Answer:
The nature of the problem is such that it is not a single bank's alone (unless it disconnects from the network). The entire network must be secured and until that happens the bank and its account holders are potentially vulnerable.

3.10.1. The Road Ahead?

It is envisaged that the process will be driven by the card associations (i.e., VISA, MasterCard, etc). This is due to their role and influence over the infrastructure. There should be a revision of the associated standards involving new design and security requirements. These requirements could be very prescriptive, limiting what functionality is allowed. Following a finalization of the standards, the vendors will then update products. Hopefully this will provide an opportunity for more uniformity and collaboration between different vendor product offerings. (This would makes business sense for institutions, given them more flexibility, choice and bargaining power). Finally, the card associations will mandate the new requirements to member institutions.

3.11. The Unanswered Question

Who is liable in the event of such an attack leading to fraud?