Security APIs: 
The last word on ATM security? 
The first word on TC?

Mike Bond
Computer Security Group
Talk Structure

• Introducing Security APIs
• Discovering Security APIs: ATM security
  – ATM Security Basics
  – Early attacks on Financial HSMs
  – Finding Faults in Type Systems
  – Problems with DES
  – Information Leakage Attacks
• The Future of Security APIs: Trusted Computing
  – “Digital Battlefields”
  – Getting formal
• Conclusions
What is a Security API?

- An API that allows users to work with sensitive data and keys, and uses cryptography to enforce a policy on the usage of data.
Who Needs Security APIs?

- Those who need to enforce access policies to sensitive information
  Example: Granting signing permission at a Certification Authority

- Those who need to protect mission critical sensitive data
  Example: Protecting PIN generation keys at banks

- Those who need to protect data in hostile environments
  Example: Protecting Token Vending Machines (Electricity, National Lottery etc…)

- Those with high crypto throughput requirements
  Example: SSL acceleration for webservers
Research into Security APIs

- Some work in early 90s using prolog style search to find attacks, but few documented attacks
- Work started in 2000 at University of Cambridge with analysis of Hardware Security Modules used in banks to protect PINs for ATMs
- New work found many more attacks, and produced first significant catalogue of API failures
- Scope has been broadened to include security modules used by certification authorities and also general purpose crypto libraries (e.g. PKCS#11, Chrysalis-ITS Luna CA3, nCipher nCore and payShield APIs)
- Latest work revisiting financial APIs examining PIN generation and verification procedures
The Simplest Security API

Plaintext → Km → Ciphertext

\[ P \rightarrow \{ P \}_{Km} \]
Protocol Notation

- Informal notation, common in textbooks

\[ A \rightarrow B : \{ X \}^{k_1}, \{ KS, A, B \}^{k_2} \]
Example Security API Commands

U→C : \{ A \}_{KM}, \{ B \}_{KM}
C→U : \{ A+B \}_{KM}

U→C : GUESS, \{ ANS \}_{KM}
C→U : YES (if GUESS=ANS else NO)

U→C : \{ X \}_{K1}, \{ K1 \}_{KM}, \{ K2 \}_{KM}
C→U : \{ X \}_{K2}
Example Type Diagram

Contains terms of the form $\{ X \}_{\text{K-TYPE-E}}$

Transaction
- $U \rightarrow C : \{ X \}_{\text{K-TYPE-D}}$
- $C \rightarrow U : \{ X \}_{\text{K-TYPE-F}}$
Hardware Security Modules

• An instantiation of a security API
• Often physically tamper-resistant (epoxy potting, temperature & x-ray sensors)
• May have hardware crypto acceleration (not so important with speed of modern PC)
• May have special ‘trusted’ peripherals (key switches, smartcard readers, key pads)

(referred to as **HSMs** subsequently)
Hardware Security Modules
Why ATM Network Security?

• ATM security was the “killer-app” that brought cryptography into the commercial mainstream – so long history of financial API development

• Concrete and simple security policy for APIs:
  “Only the customer should know her PIN.”
  “Keys protecting PINs may only be manipulated when authorised by two different employees.”

• IBM made CCA manual publicly available
  – Excellent detailed description of API
  – Good explanation of background to PIN processing APIs
  – Unfortunately: lots of uncatalogued weaknesses.
ATM Security Basics

- The crucial secret is the customer PIN. The customer should be the only person that knows the value of this PIN.
- PINs need to be protected from malicious insiders and outsiders.
- PINs must be protected when generated, in storage, when issued to customers, when travelling via the international ATM network, and when being verified.
- To this end, banks use Hardware Security Modules (HSMs) to perform cryptography and implement a policy which prevents both insiders and outsiders from gaining unauthorised access to PINs.
Security Modules in Banks
How are PINs Generated?

Start with your bank account number (PAN)

5641 8203 3428 2218

Encrypt with PIN Derivation Key
(aka PMK – Pin Master Key)

22BD 4677 F1FF 34AC

decimalise

Chop off the End

2213

(B→1)  (D→3)
What’s a Decimalisation Table?

- Remember encrypted result was in hexadecimal?
- Encryption produces output that looks uniformly distributed, so 0–F are all equally likely
- Decimalisation Table used to map 0–F back to 0–9

<table>
<thead>
<tr>
<th>digit in</th>
<th>0123456789ABCDEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit out</td>
<td>0123456789012345</td>
</tr>
</tbody>
</table>

e.g. 22BD -> 2213

- Because some numbers have several hexadecimal digits mapped to them, they are more likely to occur in issued PINs than others
Collecting Frequency Distributions
Sample size: 45 people (just large enough to prove non-uniform hypothesis with 1% conf)
How do I change my PIN?

• Most store an offset between the original derived PIN and your chosen PIN

• Example bank record…
  – PAN 5641 8233 6453 2229
  – Name Mr M K Bond
  – Balance $1234
  – PIN Offset 0000

• If I change PIN from 4426 to 1979, offset stored is 7553 (each digit is independent modulo 10)

• Some systems do work completely differently. You choose your PIN at the outset in these.
Early Attacks on Financial HSMs
XOR To Null Key Attack

- Top-level crypto keys exchanged between banks in several parts carried by separate couriers, which are recombined using the exclusive-OR function.

```
Source
HSM

KP1

KP2

Dest
HSM

Repeat twice...
User->HSM : Generate Key Component
HSM->Printer : KP1
HSM->User : \{ KP1 \}_ZCMK

Combine components...
User->HSM : \{ KP1 \}_ZCMK, \{ KP2 \}_ZCMK
HSM->User : \{ KP1 \text{ xor } KP2 \}_ZCMK

Repeat twice...
User->HSM : KP1
HSM->User : \{ KP1 \}_ZCMK

Combine components...
User->HSM : \{ KP1 \}_ZCMK, \{ KP2 \}_ZCMK
HSM->User : \{ KP1 \text{ xor } KP2 \}_ZCMK
```
XOR To Null Key Attack

- A single operator could feed in the same part twice, which cancels out to produce an ‘all zeroes’ test key. PINs could be extracted in the clear using this key.

**Combine components...**

User→HSM : \{ KP1 \}_{ZCMK}, \{ KP1 \}_{ZCMK}

HSM→User : \{ KP1 \text{ xor } KP1 \}_{ZCMK}

KP1 \text{ xor } KP1 = 0
Offset Calculation Attack

- Bank adds a new command to the API to calculate the offset between a new generated PIN and the customer’s chosen PIN.
- Possessing a bank account gives knowledge of one generated PIN. Any customer PIN could be revealed by calculating the offset between it and the known PIN.

U→C : Old PAN, Old offset, New PAN
C→U : New offset
ATMs are simpler than HSMs and have only one master key. ATMs need to be sent Terminal Communications keys (session keys) for link cryptography.

**Master Keys**
- TC – terminal communications
- TMK – terminal master keys & PIN derivation keys
- ZCMK – zone control master keys (between HSMs)
- WK – working keys (session keys)
- LP – local PIN storage key

**Master Key**
- TMK-ATM - used for everything

\[
\text{HSM} \quad \xrightarrow{TC1} \quad \text{ATM}
\]

\[
\{ \text{TC1} \} \text{TC} \quad \text{but how?} \quad \rightarrow \quad \{ \text{TC1} \} \text{TMK-ATM}
\]
Type System Attack (2)

- PIN derivation keys (PDKs) share the same type as Terminal Master Keys (TMKs), and encrypting communication keys for transfer to an ATMs uses exactly the same process as calculating a customer PIN – encryption with single DES.

\[
\begin{align*}
\text{User} \rightarrow \text{HSM} & : \text{TCl} \\
\text{HSM} \rightarrow \text{User} & : \{ \text{TCl} \}_T^C \\
\text{User} \rightarrow \text{HSM} & : \{ \text{TCl} \}_T^C , \{ \text{TMK-ATM} \}_T^M \\
\text{HSM} \rightarrow \text{User} & : \{ \text{TCl} \}_T^M-\text{ATM}
\end{align*}
\]

The attack...

\[
\begin{align*}
\text{User} \rightarrow \text{HSM} & : \text{PAN} \\
\text{HSM} \rightarrow \text{User} & : \{ \text{PAN} \}_T^C \\
\text{User} \rightarrow \text{HSM} & : \{ \text{PAN} \}_T^C , \{ \text{PDK1} \}_T^M \\
\text{HSM} \rightarrow \text{User} & : \{ \text{PAN} \}_T^{PDK1}
\end{align*}
\]
VSM Type Diagram

TMK/PIN → WK → ZCMK

ZCMK_I → TMK_I → WK_I → LP

(RAND) → TC → (CLEAR) → TC_I
Type System Attack (Graphical)
Problems With DES

- A thief walks into a car park and tries to steal a car...

- How many keys must he try?
Car Park Analogy 1900
Car Park Analogy 2000
The Meet in the Middle Attack

- Common sense statistics
- Attack multiple keys in parallel
- Need the same plaintext under each key
- Encrypt this plaintext to get a ‘test vector’
- Typical case: A $2^{56}$ search for one key becomes a $2^{40}$ search for $2^{16}$ keys
- Poor implementations of 3DES key storage allow 3DES key halves to be attacked individually
MIM Attack on DES Security Modules

- Generate $2^{16}$ keys
- Encrypt test vectors
  
  $U \rightarrow C : \{ \text{KEY1} \}_{KM}$
  
  $C \rightarrow U : \{ 0000000000000000 \}_{\text{KEY1}}$

- Do $2^{40}$ search

Cryptoprocessor’s Effort $\rightarrow$ Search Machine’s Effort

<table>
<thead>
<tr>
<th>16 bits</th>
<th>40 bits</th>
</tr>
</thead>
</table>

56 bit key space
MIM Attack on Triple-DES HSMs

\[ E_K(D_K(E_K(\text{KEY})) = E_K(\text{KEY}) \]

Single Length Key

Double Length “Replicate”

Double Length
Information Leakage Attacks

- Remember PINs derived from account numbers
- Hexadecimal raw PIN is converted to decimal using decimalisation table
- Most APIs allow the decimalisation table to be specified with each PIN verification command
- A normal verification command eliminates one of 10,000 combinations of PIN for the attacker.
- If the table is altered, whether or not the alteration affects correct verification leaks much more information about the PIN

examples...

(Bond/Clulow 2002)
Decimalisation Table Attack (1)

1. Encrypt PAN
   Raw PIN = 22BD
2. Decimalise
   Natural PIN = 2213
3. Verify
   0000 != 2213

Encrypted PMK
48CCA975F4B2C8A5

PAN
5641820334282218

Trial PIN
0000

PIN_Verify

Yes/No
(eliminates 1 combination)
Decimalisation Table Attack (2)

1. Encrypt PAN
   Raw PIN = 22BD
2. Decimalise
   Natural PIN = 0000
3. Verify
   0000 = 0000

Yes/No
(eliminates all PINs containing digit 7)

Encrypted PMK
48CCA975F4B2C8A5

PAN
5641820334282218

Trial PIN
0000

0123456789ABCDEF
0000000100000000
Decimalisation Table Attack (3)

1. Encrypt PAN
   Raw PIN = 22BD
2. Decimalise
   Natural PIN = 1100
3. Verify
   0000 != 1100

Yes/No
(shows PIN contains digit 2)
Decimalisation Table Attack (4)

Encrypted PMK
48CCA975F4B2C8A5

PAN
5641820334282218

Encrypted Trial PIN
{2213}_KM

0123456789ABCDEF
0123456789012345

1. Encrypt PAN
Raw PIN = 22BD

2. Decimalise
Natural PIN = 2213

3. Verify
2213 = 2213

Yes/No
(no information)
**Decimalisation Table Attack (5)**

1. **Encrypt PAN**
   Raw PIN = 22BD

2. **Decimalise**
   Natural PIN = 2213

3. **Verify**
   2213 = 2213

PIN _Verify_

Yes/No
(eliminates PINs containing digit 7)
PAN Modification Attack (1)

- Encrypted PINs transferred from ATM to issuing bank via ATM network using point to point encryption
- At each node PIN block must be decrypted with incoming key, and re-encrypted with outgoing key
- Common ISO standard “binds” PIN to particular customer by exclusive-ORing PAN with PIN before encryption
- Attack: specifying incorrect PAN may make deduced PIN contain hexadecimal digit ‘A’-’F’, which causes formatting error. Conditions under which formatting error arises leaks information about PIN.

(Clulow 2002)
PIN Block Formats

ISO-0

Primary Account Number (PAN)

```
5461 8203 6345 2239
```

Format ID

```
041234FFFFFFFFFFFF
```

PIN

```
0000 8203 6345 2239
```

padding

```
xor
```

```
0412A6FC9CBADDC6
```

ISO-2

```
241234FFFFFFFFFFFF
```
PAN Modification Attack (2)

\[
\text{PIN}_\text{Translate} \\
\{\text{PIN Block}\}_{\text{IWK}} \quad \{\text{AWK}\}_{\text{KM}} \quad \{\text{IWK}\}_{\text{KM}} \\
\]
PAN Modification Attack (3)

041234FFFFFFFFFFFFF
xor
000820363452239
=
0412B6FC9CBADDCC6

correct PAN removed

0412B6FC9CBADDCC6
xor
000820363452239
=
041234FFFFFFFFFFFFF

construction of PIN block

0412B6FC9CBADDCC6
xor
000720363452239
=
0412C4FFFFFFFFFFFFF

modified PAN

PIN

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td>E</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>E</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>8</td>
<td>9</td>
<td>E</td>
<td>F</td>
<td>C</td>
<td>D</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>E</td>
<td>9</td>
<td>8</td>
<td>B</td>
<td>A</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>F</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Removed – PIN contains ‘C’ – error
1. Encrypt PAN
   Raw PIN = 22BD
2. Decimalise
   Natural PIN = 2213
3. Add Offset
   PIN + 0000 = 2213
3. Format as ISO-0 PINblock
   042213FFFFFFFF
   xor
   0000820334282218
3. Encrypt block under WK
   FA28CF742A3C08A5
# Collecting Frequency Distributions

<table>
<thead>
<tr>
<th>PMK</th>
<th>PIN Block</th>
<th>Output</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D7604EBA10AC7F3</td>
<td>043328FFFFFFFE</td>
<td>FD29DA10029726DC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xor 0000820362342219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23AD73218F2C0AB1</td>
<td>049106FFFFFFFE</td>
<td>EA4118F2C0AB3AC6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xor 0000820362342219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9E760AF7F34EFA10</td>
<td>041522FFFFFFFE</td>
<td>104AE02F763A56DF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xor 0000820362342219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66F7604EB263543C</td>
<td>046467FFFFFFFE</td>
<td>2E6892FC328D5212</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xor 0000820362342219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14E247F78EA876A0</td>
<td>042315FFFFFFFE</td>
<td>5323AB35C00273BB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xor 0000820362342219</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next… assemble distribution in a loop (mod 10)
1. Encrypt PAN
   Raw PIN = 22BD
2. Decimalise
   Natural PIN = 2213
3. Add Offset
   PIN + 0000 = 2213
3. Format as ISO-0 PINblock
   042213FFFFFFFF
   xor
   0000820334282218
3. Encrypt block under WK
   FA28CF742A3C08A5
Aligning Frequency Distributions

(four alignment problems along orthogonal axes...)
The Last Word on ATM Security?

- We have come along way since the first flaws in PIN processing systems were put in the public domain.
- In Europe, this entire architecture may be on the way out, as EMV (“Chip and PIN”) is phased in.
- Could these be the last published attacks on PIN processing systems?
- Banking security is concerned as much with risk and liability as with cryptographic security – there may be more to learn in fields where cryptographic security is a higher priority.
- What next for Security API research?
The First Word on Trusted Computing?

- Trusted Computing proposals put simple hardware security modules in every PC
- TC also encourages compartmentalisation of applications into trusted and untrusted components – just like the evolution of ATM security
- Security API research may be able to help the designers of these interfaces avoid the worst mistakes, or maybe even make the interfaces secure?
A double-edged sword?

- **IRM** – Information Rights Management
  - Companies can stop leaks
  - Mafia can keep their records secret
- **DRM** – Digital Rights Management
- **Trusted IO** – Enter your ATM PIN at your PC
- **Global PKI** – All devices potentially identifiable
- **Trusted Anonymity Systems**
- **Truly Anonymous peer-to-peer systems**
- **High-availability systems**
- **Reverse-engineering resistant viruses**
Example: Information Rights Management

- Microsoft Office 2003 with Microsoft Rights Management Server
- Will it be secure when supported by TC?

The “restrict” button
“The Digital Battlefield”

Ring 2+

- App1
- App2

Ring 1

- Services
- Drivers

Ring 0

- O/S

Hardware

- Nexus

TPM / SSC
“The Digital Battlefield”

Ring 2+

- DRM App
- App2

Ring 1

- Services
- Drivers
- Nexus

Ring 0

- O/S

Hardware

- TPM / SSC
“The Digital Battlefield”

Ring 2+

- DRM App
- MyApp

Ring 1

- Services
- Drivers

Ring 0

- O/S

Hardware

- Nexus
- TPM / SSC
"The Digital Battlefield"

Ring 2+

- DRM App
- MyApp

Ring 1

- Services
- Drivers

Ring 0

- O/S (hacked)

Hardware

Nexus

TPM / SSC
“The Digital Battlefield”

Ring 2+

- DRM App
- MyApp

Ring 1

- Services
- Drivers

Ring 0

- O/S (hacked)
- Nexus

Hardware

- TPM / SSC
Getting Formal

- How are we going to survive on this ‘battlefield’ if all our technology is for attack, not defence?
- So far we only have heuristics for understanding how to design Security APIs, but there are important properties we would like to gain assurance about (in formal speak: “prove”)
- Formalising the specification of Security APIs could help make properties clearer
- Semi-automated analysis of specifications could assist gaining assurance, locating vulnerabilities, and enumerating all instances of vulnerabilities
Data_Key_Import

Data_Key_Import (CSNBDMK)

<table>
<thead>
<tr>
<th>Platform/Product</th>
<th>OS/360</th>
<th>AX</th>
<th>NT</th>
<th>OS/400</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 4758-1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

The Data_Key_Import verb imports an encrypted, source DES DATA key and creates or updates a target internal key token with the master-key-encrypted source key. The verb can import the key into an internal key token in application storage or in key storage. This verb, which is authorized with a different control point than used with the Key_Export verb, allows you to limit the export operations to DATA keys as compared to the capabilities of the more general verb.

Specify the following:

- An external key token containing the source key to be imported. The external key must indicate that a control vector is present; however, the control vector is usually valued at zero.
- Alternatively, you can provide the encrypted data key at offset 16 in an otherwise all X'00' key token. The verb will process this token format as a DATA key encrypted by the importer key and a null (all zero) control vector.
- An IMPORTER key-encrypting key under which the source key is deciphered.
- An internal or null key token. The internal key token can be located in application data storage or in key storage.

The verb builds the internal key token by the following:

- Creates a default control vector for a DATA key type in the internal key token, if the control vector in the external key token is 0. If the control vector is not zero, the verb copies the control vector into the internal key token from the external key token.
- Multiplies-encrypts the key under the keys formed by the exclusive-OR of the key-encrypting key (identified in the importer_key_identifier) and the control vector in the external key token, then multiplies-encrypts the key under keys formed by the exclusive-OR of the master key and the control vector in the internal key token. The verb places the key in the internal key token.
- Calculates a token-validation value and stores it in the internal key token.

This verb does not adjust the key parity of the source key.

Restrictions

None

Data_Key_Import

Format

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSENBDMK</td>
<td></td>
</tr>
<tr>
<td>return_code</td>
<td>Output Integer</td>
</tr>
<tr>
<td>reason_code</td>
<td>Output Integer</td>
</tr>
<tr>
<td>ext_key_length</td>
<td>InputOutput Integer</td>
</tr>
<tr>
<td>ext_data</td>
<td>InputOutput String</td>
</tr>
<tr>
<td>ext_key</td>
<td>InputOutput String</td>
</tr>
<tr>
<td>ext_data_length</td>
<td>InputOutput Integer</td>
</tr>
<tr>
<td>source_key_token</td>
<td>Input String 64 bytes</td>
</tr>
<tr>
<td>importer_key_identifier</td>
<td>Input String 64 bytes</td>
</tr>
<tr>
<td>target_key_identifier</td>
<td>InputOutput String 64 bytes</td>
</tr>
</tbody>
</table>

Parameters

For the definitions of the return_code, reason_code, ext_key_length, and ext_data parameters, see "Parameters Common to All Verbs" on page 1-10.

- **source_key_token**
  The source_key_token parameter is a pointer to a 64-byte string variable containing the source key to be imported. The source key must be an external key.

- **importer_key_identifier**
  The importer_key_identifier parameter is a pointer to a 64-byte string variable containing the (IMPORTER) transport key used to decipher the source key.

- **target_key_identifier**
  The target_key_identifier parameter is a pointer to a 64-byte string variable containing a null key token, an internal key token, or the key label of an internal key token or null key token record in key storage. The key token receives the imported key.

Required Commands

The Data_Key_Import verb requires the Data Key Import command (offset X'0109') to be enabled in the hardware.
First Steps: Theorem Proving

- Predicate $U(x)$ represents adversary knowledge; implications represent adversary gaining knowledge through transactions. Manual pages from previous slide condensed:

$$U(e(x, \text{xor}(k, t))) \land U(e(k, \text{xor}(km, \text{imp})) \rightarrow U(e(x, \text{xor}(km, t))))$$

- We assert that there is an attack, and challenge the tool to prove it

$$U(a\_secret).$$
Early Results

• Driving theorem provers is difficult – a whole new world of terminology and expertise to be learned, made more difficult because the tools are highly abstract.

• In the right hands, the tools are powerful: we can model all known “pure” API attacks (not involving properties of crypto)

• It already looks like theorem proving will be useful for enumerating all instances of a general attack method e.g. “type-casting” on the IBM 4758 CCA

• We hope to enumerate all ‘Meet-in-the-Middle’ attacks on a security API next
Conclusions

- We have learnt a lot from analysing ATM security, but there is still much much more to do…

- If and when Trusted Computing arrives on our desktops, Security APIs will not be a specialist backwater of protocol analysis, but an integral part of secure application design

- We are making the first steps to try and bring order and sense to the catalogue of attacks on existing Security APIs, and there is plenty of room for more research, which might have a real impact on the long-term success of Trusted Computing.
More Information

Papers, Links & Resources
http://www.cl.cam.ac.uk/~mkb23/research.html
http://www.cl.cam.ac.uk/~jc407/

Attacks on IBM 4758 CCA & Hardware Cracker
http://www.cl.cam.ac.uk/~rnc1/descrack/

Phantom Withdrawals and Banking Security
http://www.cl.cam.ac.uk/~mkb23/phantom/

I am around for the rest of the afternoon…

Email… Mike.Bond@cl.cam.ac.uk