Formal Analysis of Security APIs

Graham Steel

INRIA & LSV, ENS de Cachan
PIN Processing APIs

Photo: redspotted/Flickr
Verizon Breach Report 2008

Released April 2009
Verizon Breach Report 2008

Released April 2009

“While statistically not a large percentage of our overall caseload in 2008, attacks against PIN information represent individual data-theft cases having the largest aggregate exposure in terms of unique records,”

“In other words, PIN-based attacks and many of the very large compromises from the past year go hand in hand.”
Verizon Breach Report 2008

Released April 2009

“While statistically not a large percentage of our overall caseload in 2008, attacks against PIN information represent individual data-theft cases having the largest aggregate exposure in terms of unique records,”

“In other words, PIN-based attacks and many of the very large compromises from the past year go hand in hand.”

“We’re seeing entirely new attacks that a year ago were thought to be only academically possible,”
Verizon Breach Report 2008

Released April 2009

“While statistically not a large percentage of our overall caseload in 2008, attacks against PIN information represent individual data-theft cases having the largest aggregate exposure in terms of unique records,“

“In other words, PIN-based attacks and many of the very large compromises from the past year go hand in hand."

“We’re seeing entirely new attacks that a year ago were thought to be only academically possible,”

“What we see now is people going right to the source [...] and stealing the encrypted PIN blocks and using complex ways to un-encrypt the PIN blocks.”

(Quotes from Wired Magazine interview with report author, Bryan Sartin)
HSMs

- Manufacturers include IBM, VISA, nCipher, Thales, Utimaco, HP
- Cost around $10 000
Deriving a PIN: IBM 3624 Method

IPIN derived by:

Encode account number (PAN) as 0000AAAAAAAAAAAAAAA
Deriving a PIN: IBM 3624 Method

IPIN derived by:

Encode account number (PAN) as 0000AAAAAAAAAAAAA

3DES encrypt under a PDK (PIN Derivation Key)
Deriving a PIN: IBM 3624 Method

IPIN derived by:

Encode account number (PAN) as 0000FFFFFFFFFFFFFFF

3DES encrypt under a PDK (PIN Derivation Key)

Take 4 leftmost hexadecimal digits of result
Deriving a PIN: IBM 3624 Method

IPIN derived by:

Encode account number (PAN) as 0000AAAAAAAAAAAAAAA

3DES encrypt under a PDK (PIN Derivation Key)

Take 4 leftmost hexadecimal digits of result

Decimalise using a mapping table (‘dectab’)

0123456789ABCDEF
0123456789012345
Deriving a PIN: IBM 3624 Method

IPIN derived by:

Encode account number (PAN) as 0000AAAAAAAAAAAAA

3DES encrypt under a PDK (PIN Derivation Key)

Take 4 leftmost hexadecimal digits of result

Decimalise using a mapping table (‘dectab’)

0123456789ABCDEF
0123456789012345

PIN = IPIN + Offset (modulo 10 each digit)
PIN Processing API

Verify PIN:

\[\{\text{PIN}\}_K, \text{PAN}, \text{Dectab} \rightarrow \text{Offset} \]

yes/no \[\leftarrow \quad \text{K, PDK}\]
PIN Processing API

Verify PIN:

\[\{\text{PIN}\}_K, \text{PAN}, \text{Dectab} \rightarrow \]

Offset

yes/no

If host machine is attacked, PIN should remain secure (ANSI X7.8, ISO 9564 requirement)
Decimalisation Table Attack (Clulow ’02, Bond & Zeilinski ’03)

Suppose in a hacked switch, an attacker has a set 
\{PIN\}_K, PAN, Dectab, Offset that verifies PIN is correct
Decimalisaton Table Attack (Clulow ’02, Bond & Zeilinski ’03)

Suppose in a hacked switch, an attacker has a set 
\{PIN\}_K, PAN, Dectab, Offset that verifies PIN is correct

Original Dectab

0123456789ABCDEF
0123456789012345

Dectab’

0123456789ABCDEF
1123456789112345
Decimalisation Table Attack (Clulow ’02, Bond & Zeilinski ’03)

Suppose in a hacked switch, an attacker has a set \( \{\text{PIN}\} \cup \{\text{PAN}, \text{Dectab}, \text{Offset}\} \) that verifies PIN is correct.

Original Dectab

\[
\begin{align*}
0123456789ABCDEF \\
0123456789012345
\end{align*}
\]

Dectab’

\[
\begin{align*}
0123456789ABCDEF \\
1123456789112345
\end{align*}
\]

Repeat verification command with Dectab’

Successful verification indicates no 0s in PIN.
### More dectab attack

To find the 0s, try changing the offset

<table>
<thead>
<tr>
<th>Attacker set offset</th>
<th>Result from HSM</th>
<th>Knowledge of PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Incorrect PIN</td>
<td>????</td>
</tr>
<tr>
<td>0010</td>
<td>Incorrect PIN</td>
<td>????</td>
</tr>
<tr>
<td>0100</td>
<td>Incorrect PIN</td>
<td>????</td>
</tr>
<tr>
<td>1000</td>
<td>Incorrect PIN</td>
<td>????</td>
</tr>
<tr>
<td>0011</td>
<td>Incorrect PIN</td>
<td>????</td>
</tr>
<tr>
<td>0101</td>
<td>Correct PIN</td>
<td>?0?0</td>
</tr>
</tbody>
</table>
AnaBlock (TCS 2006)

Take a customer configuration and an API spec. as input
AnaBlock (TCS 2006)

Take a customer configuration and an API spec. as input

Using CLP, generate tree of all possible attacks
AnaBlock (TCS 2006)

Take a customer configuration and an API spec. as input

Using CLP, generate tree of all possible attacks

Meta-logical predicates allow us to calculate transition probabilities
AnaBlock (TCS 2006)

Take a customer configuration and an API spec. as input

Using CLP, generate tree of all possible attacks

Meta-logical predicates allow us to calculate transition probabilities

Apply PRISM (Kwiatkowska et. al, 2004)

Get minimum expected number of steps to determine PIN
**AnaBlock (TCS 2006)**

Take a customer configuration and an API spec. as input

Using CLP, generate tree of all possible attacks

Meta-logical predicates allow us to calculate transition probabilities

Apply PRISM (Kwiatkowska et. al, 2004)

Get minimum expected number of steps to determine PIN

Generate tree for best attack
# Attack Trees

```
0.8
XOR 2 against A1
Call Translate

0.2

P3 in 0..7

0.6
XOR 8 against A1
Call Translate

0.4

P3 in 0..8..9

0.4

P3 in 0..1..8..9

0.6
XOR 10 against A1
Call Translate

0.4

P3 in 0..2..7

0.6

P3 in 0..1..4..7

P3 in 0..9
```
## Results from AnaBlock

<table>
<thead>
<tr>
<th>No.</th>
<th>Attack</th>
<th>( E(Steps) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>ISO-0 (extended)</td>
<td>13.6</td>
</tr>
<tr>
<td>(2)</td>
<td>Dectab</td>
<td>16.145</td>
</tr>
<tr>
<td>(3)</td>
<td>Dectab &amp; ISO (restricted)</td>
<td>15.275</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>Attack</th>
<th>Range: 400</th>
<th>36</th>
<th>24</th>
<th>14</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4)</td>
<td>ISO-0 (restricted)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(5)</td>
<td>Dectab no offset</td>
<td>1</td>
<td>1</td>
<td>0.568</td>
<td>0.064</td>
<td>0.001</td>
</tr>
<tr>
<td>(6)</td>
<td>Dectab no offset</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>&amp; ISO-0 (restricted)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance of Dectab attack without offset

- Dectab without ISO-0
- Dectab with ISO

Probability vs. No. of Possible PINs
More PIN Cracking Attacks

- Dectab attacks
- Reformatting attacks
- Check value attack
- Calculate offset attack
- Competing verification algorithms attack

All require attacker to make ‘tweaked’ queries to HSM
Theory Behind Fix

Language based security
Theory Behind Fix

Language based security

- Multilevel view - high and low security
Theory Behind Fix

Language based security

- Multilevel view - high and low security
- Non-interference - no ‘flow’ from high to low
Theory Behind Fix

Language based security

- Multilevel view - high and low security
- Non-interference - no ‘flow’ from high to low
- Declassification - wrt a policy
Theory Behind Fix

Language based security

- Multilevel view - high and low security
- Non-interference - no ‘flow’ from high to low
- Declassification - wrt a policy
- Robustness - introduces integrity
Theory Behind Fix

Language based security

- Multilevel view - high and low security
- Non-interference - no ‘flow’ from high to low
- Declassification - wrt a policy
- Robustness - introduces integrity
- Endorsement - allows integrity to be raised
Theory Behind Fix

Language based security

- Multilevel view - high and low security
- Non-interference - no ‘flow’ from high to low
- Declassification - wrt a policy
- Robustness - introduces integrity
- Endorsement - allows integrity to be raised

We introduce cryptographically assured endorsement (ESORICS ’09) using MAC, and a ‘low cost’ version (NordSec ’09)
More PIN Processing

Wired Magazine, *PIN Crackers Nab Holy Grail of Bank Card Security*


M. Centenaro, R. Focardi, F. L. Luccio and G. Steel. *Type-based Analysis of PIN Processing APIs*. In ESORICS’09, LNCS 5789

Host machine

n1

k1
A(n1)

n2

k2
A(n2)

PKCS #11
Key Generate:

\[ \text{new } n,k \rightarrow h(n,k); L \]

Where \( L = \neg \text{extractable}(n), \neg \text{wrap}(n), \neg \text{unwrap}(n), \neg \text{encrypt}(n), \neg \text{decrypt}(n), \neg \text{sensitive}(n) \)
Key Management - 2

Wrap:
\[ h(x_1, y_1), h(x_2, y_2); \text{wrap}(x_1), \quad \rightarrow \quad \{y_2\}_{y_1} \]
\[ \text{extract}(x_2) \]

Unwrap:
\[ h(x_2, y_2), \{y_1\}_{y_2}; \text{unwrap}(x_2) \xrightarrow{\text{new } n_1} h(n_1, y_1); \text{extract}(n_1), \ L \]

where \( L = \)
\[ \neg \text{wrap}(n_1), \neg \text{unwrap}(n_1), \neg \text{encrypt}(n_1), \neg \text{decrypt}(n_1), \neg \text{sensitive}(n_1). \]
Host machine

\[ n_1 \]

\{k_1\}_k_2

\[ n_2 \]

Trusted device

\[ k_1 \quad x \]

\[ k_2 \quad w \]

PKCS #11
Key Management - 3

Set_Wrap: \quad h(x_1, y_1); \neg \text{wrap}(x_1) \rightarrow \text{;wrap}(x_1)

Set_Encrypt: \quad h(x_1, y_1); \neg \text{encrypt}(x_1) \rightarrow \text{;encrypt}(x_1)

\vdots

UnSet_Wrap: \quad h(x_1, y_1); \text{wrap}(x_1) \rightarrow \text{;\neg \text{wrap}(x_1)}

UnSet_Encrypt: \quad h(x_1, y_1); \text{encrypt}(x_1) \rightarrow \text{;\neg \text{encrypt}(x_1)}

\vdots

Some restrictions, e.g. can’t unset sensitive
Key Usage

Encrypt:
\[ h(x_1, y_1), y_2; \text{encrypt}(x_1) \rightarrow \{y_2\}_{y_1} \]

Decrypt:
\[ h(x_1, y_1), \{y_2\}_{y_1}; \text{decrypt}(x_1) \rightarrow y_2 \]
Key Separation Attack (Clulow, 2003)

**Intruder knows**: \( h(n_1, k_1), h(n_2, k_2) \).

**State**: \( \text{wrap}(n_2), \text{decrypt}(n_2), \text{sensitive}(n_1), \text{extract}(n_1) \)

**Wrap**: \( h(n_2, k_2), h(n_1, k_1) \rightarrow \{k_1\}_{k_2} \)

**Decrypt**: \( h(n_2, k_2), \{k_1\}_{k_2} \rightarrow k_1 \)
Host machine

n1 → k1 | x,s

n2 → k2 | w,d

{k1}k2

k1

PKCS #11
Re-import attack (DKS, 08)

**Intruder knows:** $h(n_1, k_1), h(n_2, k_2), k_3$

**State:** sensitive($n_1$), extract($n_1$), extract($n_2$)

- **Set_wrap:** $h(n_1, k_1) \rightarrow ; \text{wrap}(n_1)$
- **Set_wrap:** $h(n_2, k_2) \rightarrow ; \text{wrap}(n_2)$
- **Wrap:** $h(n_1, k_1), h(n_2, k_2) \rightarrow \{k_2\}_{k_1}$
- **Set_unwrap:** $h(n_1, k_1) \rightarrow ; \text{unwrap}(n_1)$
- **Unwrap:** $h(n_1, k_1), \{k_2\}_{k_1} \rightarrow _{\text{new } n_3} h(n_3, k_2)$
- **Wrap:** $h(n_2, k_2), h(n_1, k_1) \rightarrow \{k_1\}_{k_2}$
- **Set_decrypt:** $h(n_3, k_2) \rightarrow ; \text{decrypt}(n_3)$
- **Decrypt:** $h(n_3, k_2), \{k_1\}_{k_2} \rightarrow k_1$
Host machine

n1  \rightarrow k1  \quad w,u,x

n2  \rightarrow k2  \quad x,w

\{k2\}_{k1}

n3  \rightarrow k2  \quad x,d

PKCS #11
Two kinds of problem

- A bad ‘attribute policy’
  One can set conflicting attributes for a key

- Policy not enforced
  By copying the key using wrap/unwrap, can ‘escape’ the policy
Two kinds of problem

- A bad ‘attribute policy’
  One can set conflicting attributes for a key

- Policy not enforced
  By copying the key using wrap/unwrap, can ‘escape’ the policy

Attack this problem by first formalising ‘attribute policy’
KeyGenerate: \[ \text{new } n_1, k_1 \rightarrow h(n_1, k_1); L(n_1), \neg\text{extract}(n_1) \]

Wrap:
\[ h(x_1, y_1), h(x_2, y_2); \text{wrap}(x_1), \text{extract}(x_2) \rightarrow \{y_2\}_{y_1} \]

Unwrap:
\[ h(x_2, y_2), \{y_1\}_{y_2}; \text{unwrap}(x_2) \rightarrow h(n_1, y_1); L(n_1) \]

Encrypt:
\[ h(x_1, y_1), y_2; \text{encrypt}(x_1) \rightarrow \{y_2\}_{y_1} \]

Decrypt:
\[ h(x_1, y_1), \{y_2\}_{y_1}; \text{decrypt}(x_1) \rightarrow y_2 \]

Set_Encrypt:
\[ h(x_1, y_1); \neg\text{encrypt}(x_1) \rightarrow \text{encrypt}(x_1) \]

UnSet_Encrypt:
\[ h(x_1, y_1); \text{encrypt}(x_1) \rightarrow \neg\text{encrypt}(x_1) \]
KeyGenerate: $\xrightarrow{\text{new } n_1, k_1} h(n_1, k_1); A(n_1)$

Wrap: $h(x_1, y_1), h(x_2, y_2); \text{wrap}(x_1), \text{extract}(x_2) \rightarrow \{y_2\}_{y_1}$

Unwrap: $h(x_2, y_2), \{y_1\}_{y_2}; \text{unwrap}(x_2) \xrightarrow{\text{new } n_1} h(n_1, y_1); A(n_1)$

Encrypt: $h(x_1, y_1), y_2; \text{encrypt}(x_1) \rightarrow \{y_2\}_{y_1}$

Decrypt: $h(x_1, y_1), \{y_2\}_{y_1}; \text{decrypt}(x_1) \rightarrow y_2$

Set.Attribute.Value: $h(x_1, y_1); A_1(x_1) \rightarrow A_2(x_1)$
Attribute Policy

An \textit{attribute policy} is a finite directed graph $P = (S_P, \rightarrow_P)$ where $S_P$ is the set of allowable object states, and $\rightarrow_P \subseteq S_P \times S_P$ is the set of allowable transitions between the object states.
Attribute Policy

An *attribute policy* is a finite directed graph $P = (S_P, \rightarrow_P)$ where $S_P$ is the set of allowable object states, and $\rightarrow_P \subseteq S_P \times S_P$ is the set of allowable transitions between the object states.

An attribute policy $P = (S, \rightarrow)$ is *complete* if $P$ consists of a collection of disjoint, disconnected cliques, and for each clique $C$,

$$c_0, c_1 \in C \Rightarrow c_0 \cup c_1 \in C$$
Attribute Policy

An *attribute policy* is a finite directed graph $P = (S_P, \rightarrow_P)$ where $S_P$ is the set of allowable object states, and $\rightarrow_P \subseteq S_P \times S_P$ is the set of allowable transitions between the object states.

An attribute policy $P = (S, \rightarrow)$ is *complete* if $P$ consists of a collection of disjoint, disconnected cliques, and for each clique $C$, $c_0, c_1 \in C \Rightarrow c_0 \cup c_1 \in C$

We insist on complete policies, assuming intruder can always copy keys.
Endpoints

We call the object states of $S$ that are maximal in $S$ with respect to set inclusion *end points* of $P$.

Theorem: Derivation in API with complete policy iff derivation in API with (static) endpoint policy
Bounds

Assume endpoint policies

Make series of simple transformations
Bounds

Assume endpoint policies

Make series of simple transformations

- Bound number of fresh keys to number of endpoints $\#_{ep}$
  - get the same key every time a particular endpoint is requested
Bounds

Assume endpoint policies

Make series of simple transformations

- Bound number of fresh keys to number of endpoints \( \#ep \)
  - get the same key every time a particular endpoint is requested

- Bound number of handles to \((\#ep)^2\)
  - for each key, get one handle for each endpoint
Bounds

Assume endpoint policies

Make series of simple transformations

- Bound number of fresh keys to number of endpoints \( #_{ep} \)
  - get the same key every time a particular endpoint is requested

- Bound number of handles to \((#_{ep})^2\)
  - for each key, get one handle for each endpoint

Intruder always starts with his own key

so require \( #_{ep} + 1 \) keys and \((#_{ep} + 1)^2\) handles
KeyGenerate: \[ \new n_1, k_1 \rightarrow h(n_1, k_1); A(n_1) \]

Wrap:
\[ h(x_1, y_1), h(x_2, y_2); \ \text{wrap}(x_1), A(x_2) \rightarrow \new m_k \rightarrow \text{enc}(y_2, y_1), \text{enc}(m_k, y_1) \]
\[ \text{hmac}_{m_k}(y_2, A) \]

Unwrap:
\[ h(x_2, y_2), \text{enc}(y_1, y_2), \text{enc}(x_m, y_2), \rightarrow \new n_1 \rightarrow h(n_1, y_1); A(n_1) \]
\[ \text{hmac}_{x_m}(y_1, A); \ \text{unwrap}(x_2) \]

Encrypt:
\[ h(x_1, y_1), y_2; \ \text{encrypt}(x_1) \rightarrow \text{enc}(y_2, y_1) \]

Decrypt:
\[ h(x_1, y_1), \text{enc}(y_2, y_1); \ \text{decrypt}(x_1) \rightarrow y_2 \]

\[ P = (\{e, d, ed, w, u, wu\}, \rightarrow) \text{ (where } \rightarrow \text{ makes the obvious cliques)} \]
Model checking

We use SATMC from the AVISPA project.

Why?

- Can customize sort theory
- Can have protocols with loops
  - recent work by Roberto Carbone to detect fixpoints
- Good performance on previous API experiments
Model checking - 2

A *known key* is a key $k$ such that the intruder knows the plaintext value $k$ and the intruder has a handle $h(n, k)$.

**Property 1** If an intruder starts with no known keys, he cannot obtain any known keys.

Verified for our API in 0.4 sec
Model checking - 2

A *known key* is a key $k$ such that the intruder knows the plaintext value $k$ and the intruder has a handle $h(n, k)$.

**Property 1** If an intruder starts with no known keys, he cannot obtain any known keys.

Verified for our API in 0.4 sec

**Property 2** If an intruder starts with a known key $k_i$ with handle $h(n_i, k_i)$, and $ed(n_i)$ is true, then he cannot obtain any further known keys.

Attack
Lost session key attack

**Initial knowledge:** Handles $h(n_1, k_1)$, $h(n_2, k_2)$, and $h(n_i, k_i)$. Key $k_i$. Attributes $\text{ed}(n_1)$, $\text{wu}(n_2)$, $\text{ed}(n_i)$.

**Trace:**

Wrap: (ed) \[ h(n_2, k_2), h(n_i, k_i) \rightarrow \{k_i\}_{k_2}, \{k_3\}_{k_2}, \text{hmac}_{k_3}(k_i, \text{ed}) \]

Unwrap: (wu) \[ h(n_2, k_2), \{k_i\}_{k_2}, \{k_i\}_{k_2}, \text{hmac}_{k_i}(k_i, \text{wu}) \rightarrow h(n_2, k_i) \]

Wrap: (ed) \[ h(n_2, k_i), h(n_1, k_1) \rightarrow \{k_1\}_{k_i}, \{k_3\}_{k_i}, \text{hmac}_{k_3}(k_1, \text{ed}) \]

Decrypt: \[ k_i, \{k_1\}_{k_i} \rightarrow k_1 \]
Revised API

Wrap:
\[ h(x_1, y_1), h(x_2, y_2); \ \text{wrap}(x_1), A(x_2) \xrightarrow{\text{new } m_k} \text{enc}(y_2, y_1), \text{enc}(m_k, y_1) \]
\[ \text{hmac}_{m_k}(y_2, \ A, y_1) \]

Unwrap:
\[ h(x_2, y_2), \text{enc}(y_1, y_2), \text{enc}(x_m, y_2), \xrightarrow{\text{new } n_1} h(n_1, y_1); \ A(n_1) \]
\[ \text{hmac}_{x_m}(y_1, \ A, y_2); \ \text{unwrap}(x_2) \]

Property 2 now verified by SATMC

Can also verify attribute policy is enforced
More Key Management APIs


V. Cortier and G. Steel. *A Generic API for Symmetric Key Management*. In ESORICS ’09.

S. Fröschle and G. Steel. *Analysis of PKCS#11 APIs with Unbounded Fresh Data*, ARSPA-WITS ’09.