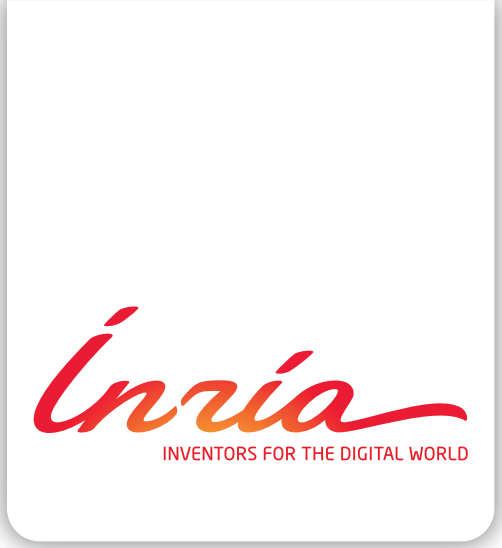


Cryptographic Key Management APIs

Graham Steel

In this Lecture

- What is a Cryptographic Security API?
- RSA PKCS#11 (Cryptoki)
- Vulnerabilities and mitigations
- Formal Analysis
- Other Crypto APIs

Cryptographic Key Management

"Key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system."

- B. Schneier, *Applied Cryptography* (2nd edition)

Management of whole key lifecycle:

- Key creation and destruction
- Key establishment and distribution
- Key storage and backup/restore
- Key use according to policy and auditing/reporting
- Key update/refresh

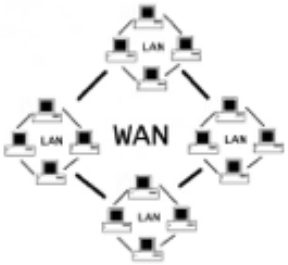
Crypto in Enterprises

CRM

Ecommerce

Enterprise portals

VPN



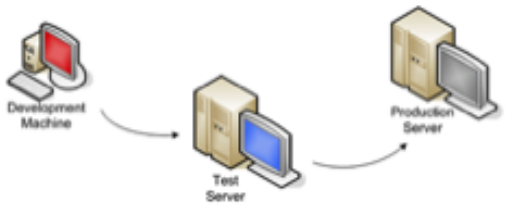
LAN and WAN

email

Databases

Backups

Business Analytics



Development / Testing

Key Management APIs

Increasing trend towards dedicated key management devices:

- HSMs for back office
- Smartcards + USB keys for agents
- Servers providing 'cryptography service' over network

Use of secure hardware for key management mandated in some sectors (e.g. in ISO 9564 for financial)

Each device has a security API

RSA PKCS#11

Public Key Cryptography Standard number 11

– PKCS are edited by the RSA company

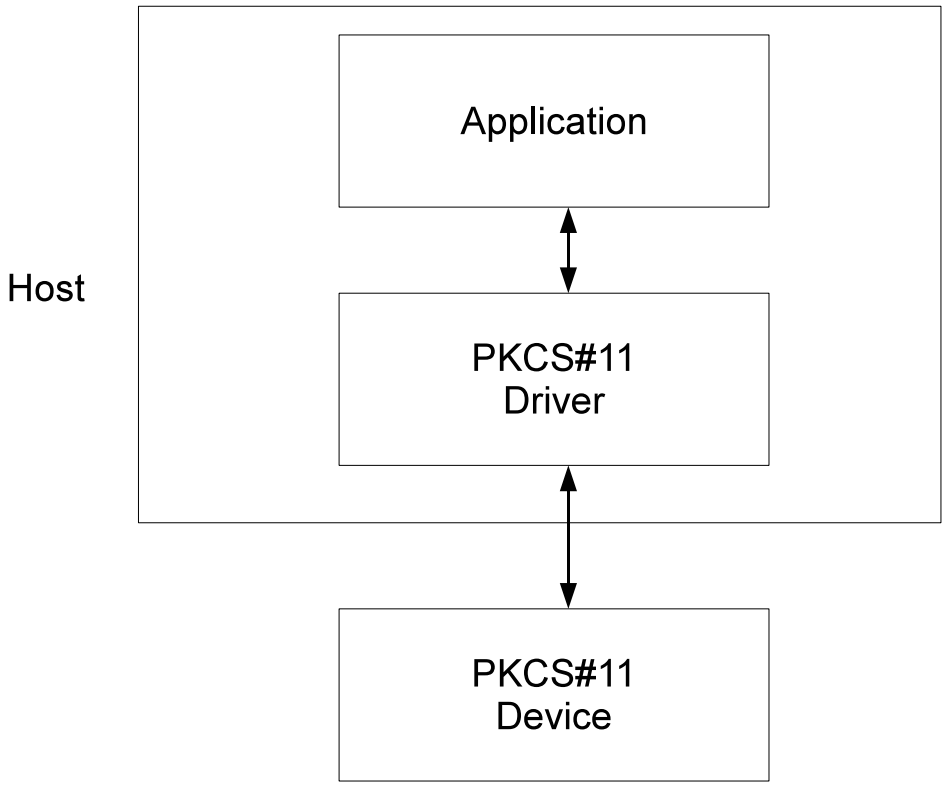
– PKCS #1 describes the RSA encryption algorithm, padding etc.

– other standards describe password based encryption, certificate formats etc.

– v1.0 of PKCS#11 1995, v2.20 2004

Browse it online:
`http://www.cryptsoft.com/pkcs11doc/v220/`

PKCS#11 Drivers



Sots, Tokens, Sessions, PINs

A PKCS#11 driver offers several *slots*

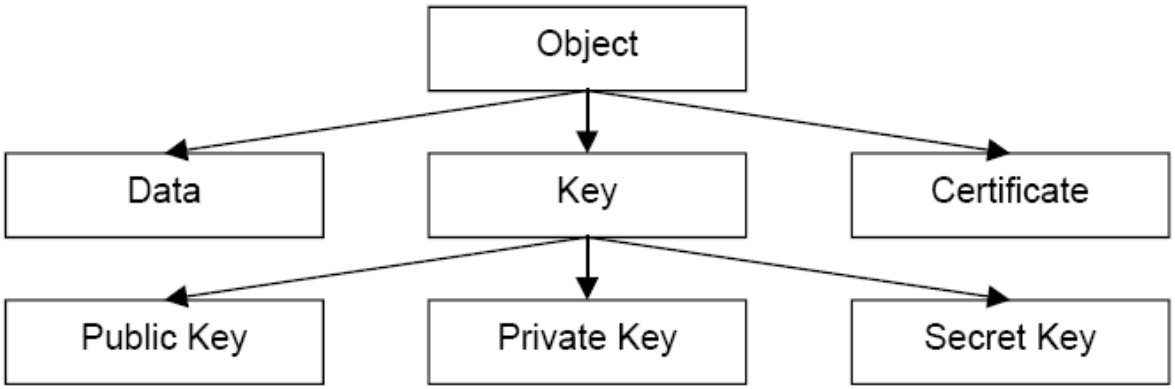
Each slot may contain a *token* (i.e. a device)

Application programs open a *session* with a token

Opening a session requires a *PIN*

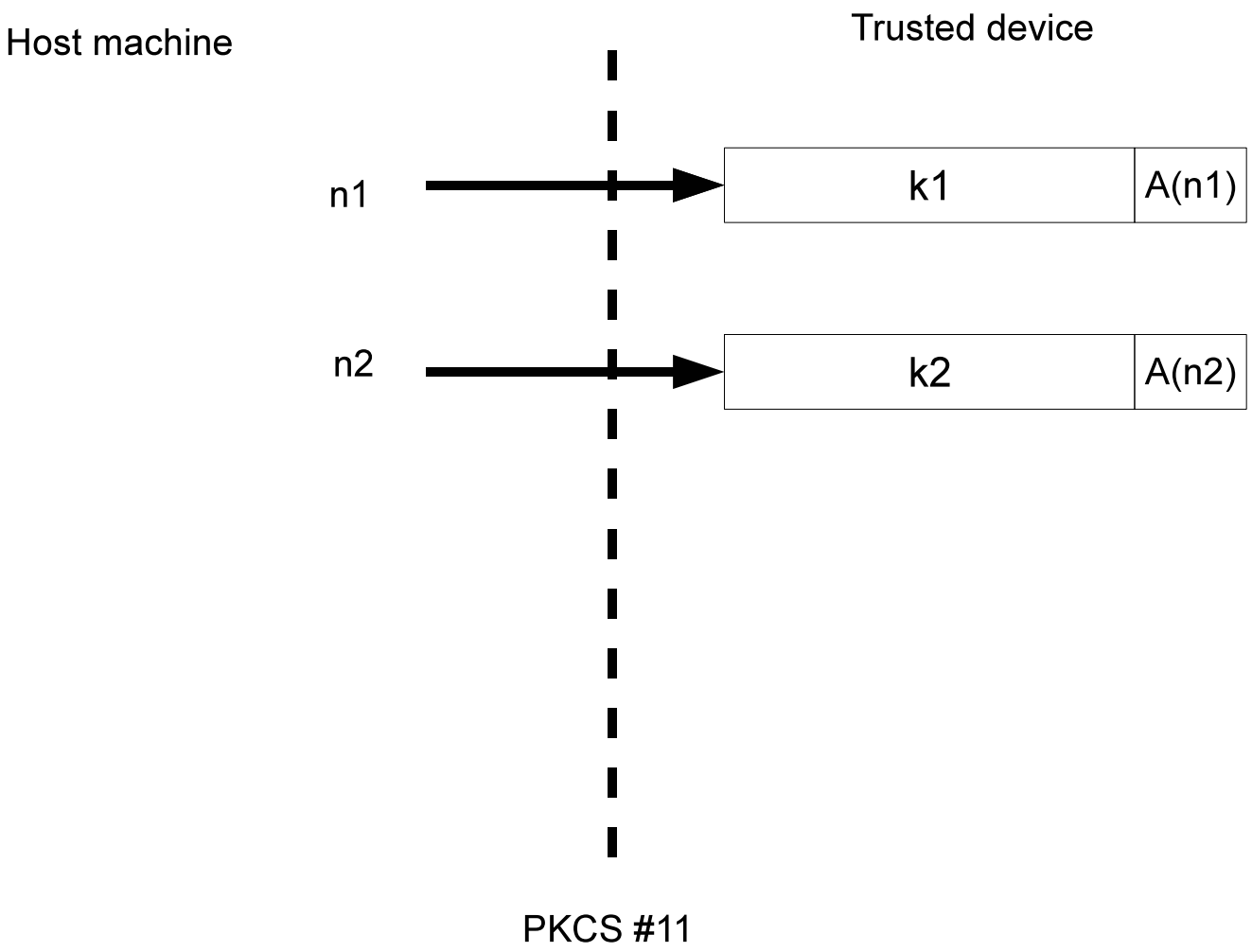
There are 2 PINs: User and Security Officer (SO)

Object Model



Keys (and other objects) accessed by *handles*

Key have *attributes* to control usage


Host machine
Trusted device
n1
k1
A(n1)
n2
k2
A(n2)
PKCS #11

Key Templates

A key template is a partial specification of key attributes

Used for creating, manipulating, and searching for objects

Consists of an array of `CK_ATTRIBUTE`s

Each attribute is a structure containing type, value, length

Order is unimportant

Generating keys

C_GenerateKey :

$$\mathcal{T} \xrightarrow{\text{new n,k}} \mathsf{h}(\mathsf{n},\mathsf{k});\mathsf{T}$$

Handle h(n,k) is returned, or `CKR_TEMPLATE_INCONSISTENT`, `CKR_TEMPLATE_INCOMPLETE`

Setting Key Attributes

$$\text{C\_SetAttributeValue} : \mathcal{T}, \mathsf{h}(\mathsf{n}, \mathsf{k}) \rightarrow \mathsf{h}(\mathsf{n}, \mathsf{k}); \mathsf{T}$$

$\mathcal{T}$ can specify new values for any attributes, but may cause `CKR_TEMPLATE_INCONSISTENT`, `CKR_ATTRIBUTE_READ_ONLY`

Wrap and Unwrap

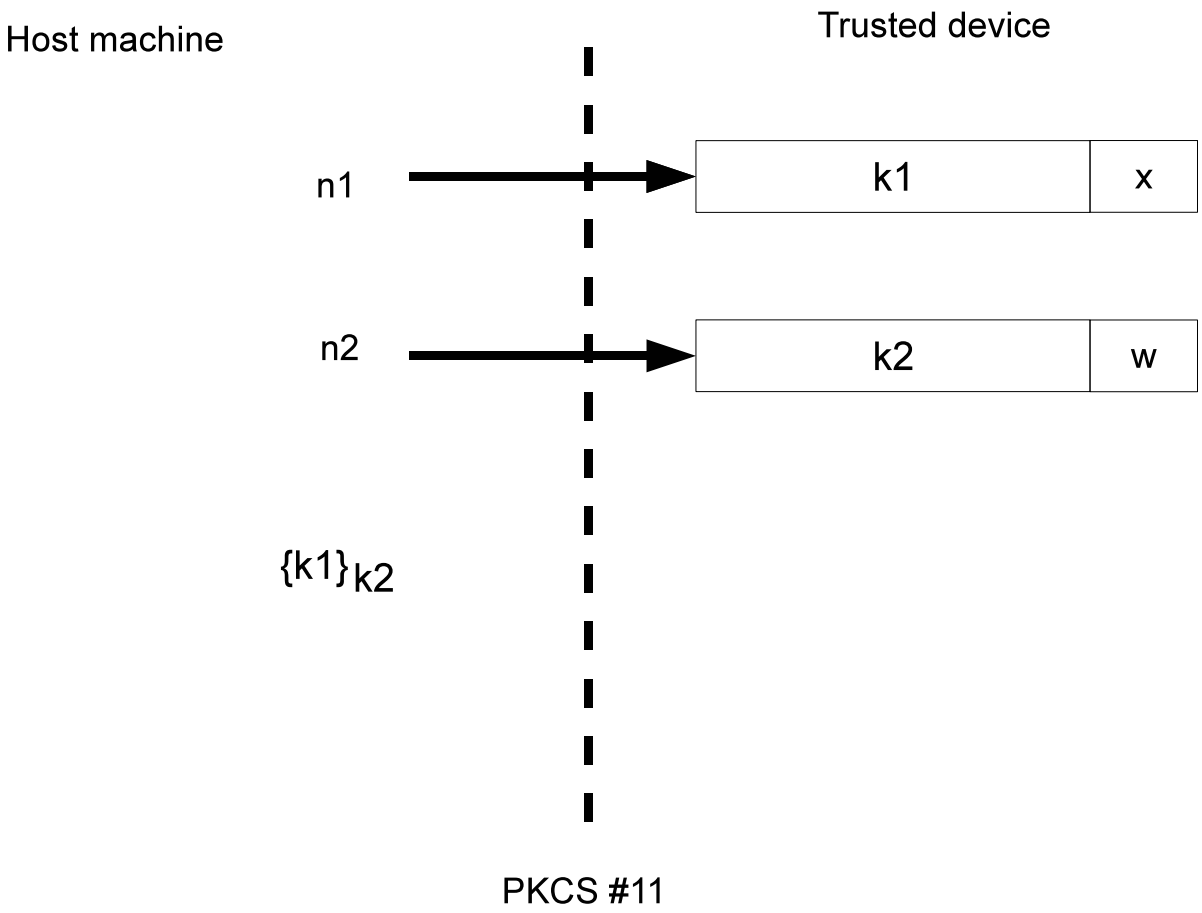
Wrap :

$$h(x_1, y_1), h(x_2, y_2);\ \mathsf{wrap}(x_1), \mathsf{extract}(x_2) \quad \rightarrow \quad \{y_2\}_{y_1}$$

Unwrap :

$$h(x_2, y_2), \{y_1\}_{y_2}, \mathcal{T};\ \mathsf{unwrap}(x_2) \xrightarrow{\text{new } n_1} h(n_1, y_1);\ \mathsf{extract}(n_1),\ T$$

May cause `CKR_WRAPPED_KEY_INVALID`, `CKR_WRAPPING_KEY_HANDLE_INVALID`, `CKR_UNWRAPPING_KEY_HANDLE_INVALID`


Host machine
Trusted device
n1
k1
x
n2
k2
w
{k1}k2
PKCS #11

Key Usage

Encrypt :

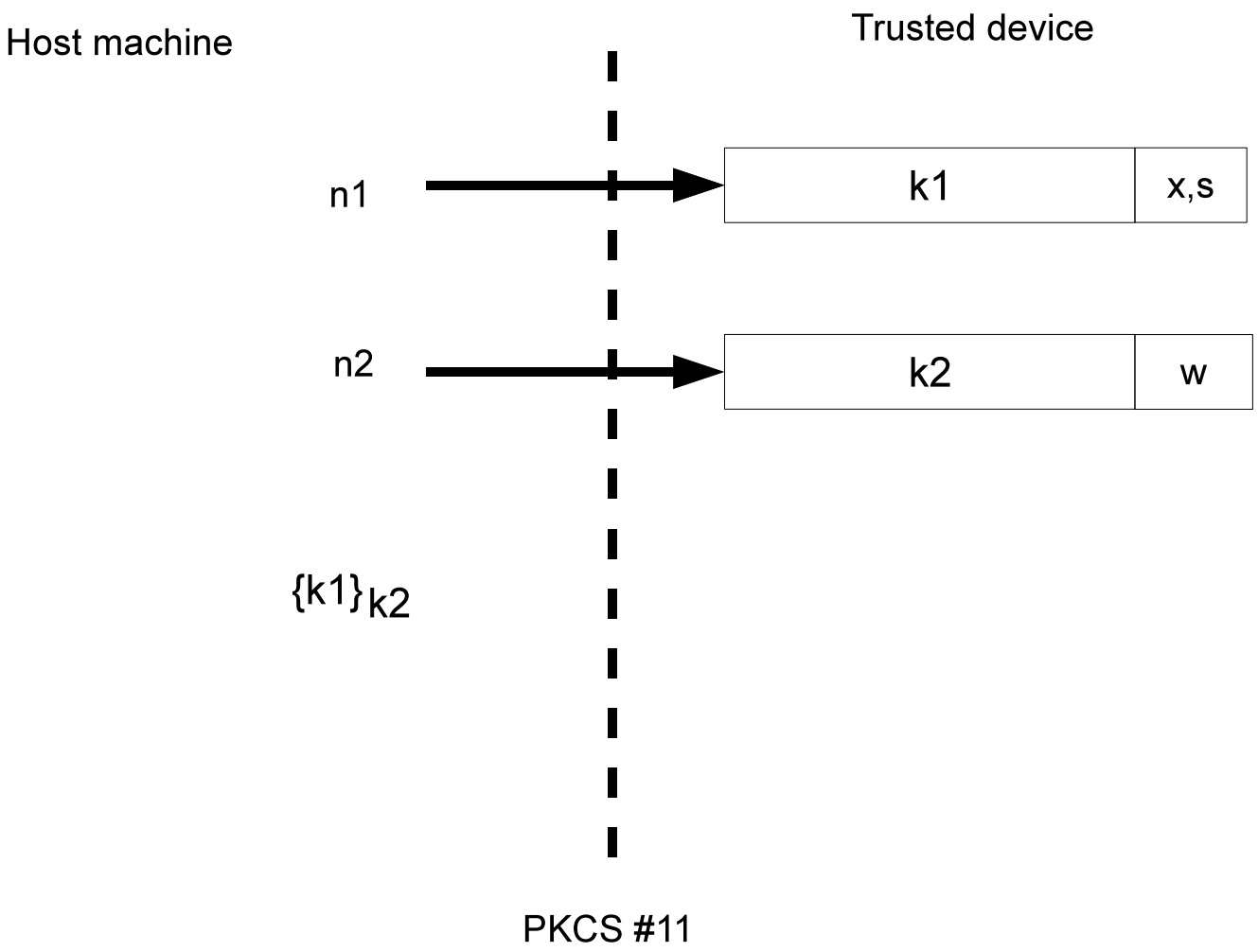
$$h(x_1, y_1), y_2;\ \text{encrypt}(x_1) \quad \rightarrow \quad \{y_2\}_{y_1}$$

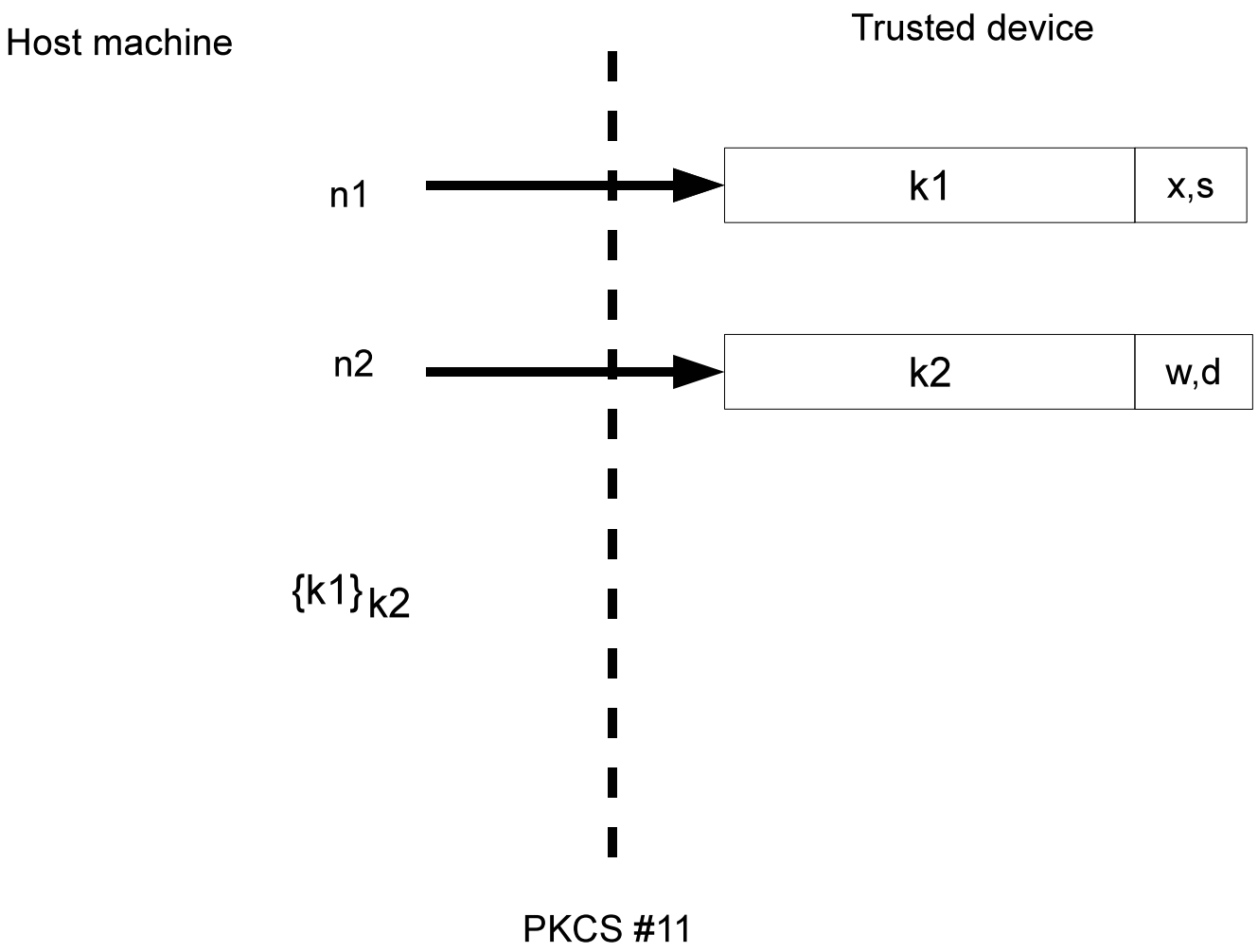
Decrypt :

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

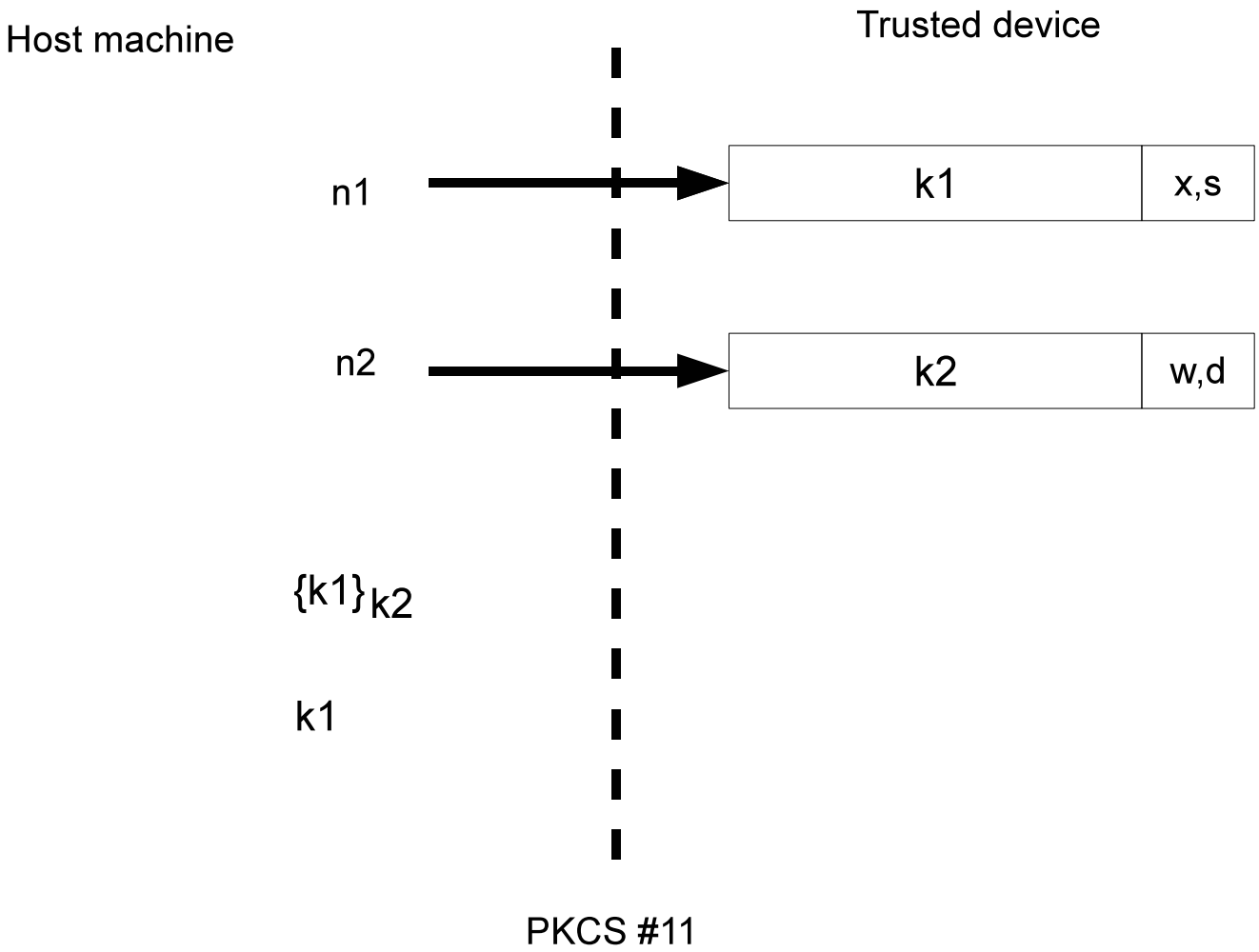
PKCS#11 Security

Section 7 of standard:
"1. Access to private objects on the token, and possibly to cryptographic functions and/or certificates on the token as well, requires a PIN.
2. Additional protection can be given to private keys and secret keys by marking them as "sensitive" or "unextractable". Sensitive keys cannot be revealed in plaintext off the token, and unextractable keys cannot be revealed off the token even when encrypted"
"Rogue applications and devices may also change the commands sent to the cryptographic device to obtain services other than what the application requested [but cannot] compromise keys marked "sensitive," since a key that is sensitive will always remain sensitive. Similarly, a key that is unextractable cannot be modified to be extractable."


Host machine
Trusted device
n1
k1
x,s
n2
k2
w
{k1}k2
PKCS #11


Host machine
Trusted device
n1
k1
x,s
n2
k2
w,d
{k1}k2
PKCS #11

Wrap/Decrypt Attack (Clulow, 2003)



Wrap/Decrypt 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: $h(n_2, k_2),\ \{k_1\}_{k_2} \rightarrow k_1$

How to fix decrypt/wrap attack?

Would like to prevent any key from being able to wrap and decrypt.

First idea: prevent any template $\mathcal{T}$ from having wrap and decrypt set in all operations (Generate, SetAttribute, Unwrap etc.)

Is this sufficient?

no..

Attack 2

Initial state: n_2 has only wrap set

Wrap:	$h(n_2, k_2)$, $h(n_1, k_1)$	$\rightarrow$	$\{k_1\}_{k_2}$
SetAttribute	$h(n_2, k_2)$, wrap=false	$\rightarrow$	$h(n_2, k_2)$; $\neg$wrap(n_2)
SetAttribute	$h(n_2, k_2)$, decrypt=true	$\rightarrow$	$h(n_2, k_2)$; decrypt(n_2)
Decrypt:	$h(n_2, k_2)$, $\{k_1\}_{k_2}$	$\rightarrow$	k_1

Sticky Attributes

Introduce some attributes which once set, cannot be unset.

Sticky_on and Sticky_off

Note: PKCS#11 already specifies some attributes like this, such as `CKA_EXTRACTABLE` and `CKA_SENSITIVE`

We add decrypt and wrap to sticky_on - is this enough?

We turn to formal methods to find out

Formal Model

[Delaune, Kremer ,S. CSF '08]

Rules:

$$T; L \xrightarrow{\text{new } \tilde{n}} T'; L'$$

Standard transition system semantics, but no longer monotonic
Attributes in the model:
encrypt, decrypt, wrap, unwrap, sensitive, extractable

Modes

$$
\begin{aligned}
h &: \text{Nonce} \times \text{Key} \rightarrow \text{Handle} \\
\text{senc} &: \text{Key} \times \text{Key} \rightarrow \text{Cipher} \\
\text{aenc} &: \text{Key} \times \text{Key} \rightarrow \text{Cipher} \\
\text{pub} &: \text{Seed} \rightarrow \text{Key} \\
\text{priv} &: \text{Seed} \rightarrow \text{Key} \\
a &: \text{Nonce} \rightarrow \text{Attribute} \quad \text{for all } a \in \mathcal{A} \\
x_1, x_2, n_1, n_2 &: \text{Nonce} \\
y_1, y_2, k_1, k_2 &: \text{Key} \\
z, s &: \text{Seed}
\end{aligned}
$$

Modelling the Attribute Policy

$$
\begin{array}{lrcl}
\text{Set\_Wrap}: & h(x_1, y_1);\ \neg\text{wrap}(x_1) & \rightarrow & \text{wrap}(x_1) \\
\text{Set\_Encrypt}: & h(x_1, y_1);\ \neg\text{encrypt}(x_1) & \rightarrow & \text{encrypt}(x_1) \\
\vdots & & \vdots & \\
\text{UnSet\_Wrap}: & h(x_1, y_1);\ \text{wrap}(x_1) & \rightarrow & \neg\text{wrap}(x_1) \\
\text{UnSet\_Encrypt}: & h(x_1, y_1);\ \text{encrypt}(x_1) & \rightarrow & \neg\text{encrypt}(x_1) \\
\vdots & & \vdots &
\end{array}
$$

Remove rules for sticky_on and sticky_off attributes

Fix decrypt/wrap attack..

Add conflicts:

$$\begin{array}{lllll} \text{Set\_Wrap}: & h(x_1, y_1);\ \neg\text{wrap}(x_1), \neg\text{decrypt}(x_1) & \rightarrow & \text{wrap}(x_1) \\ \text{Set\_Decrypt}: & h(x_1, y_1);\ \neg\text{wrap}(x_1), \neg\text{decrypt}(x_1) & \rightarrow & \text{decrypt}(x_1) \end{array}$$

Add sticky attributes:
Remove Unset_Wrap
Remove Unset_Decrypt

Another Attack

Intruder knows: $h(n_1, k_1)$, $h(n_2, k_2)$, k_3
State: $\text{sensitive}(n_1), \text{extract}(n_1)$, $\text{unwrap}(n_2)$, $\text{encrypt}(n_2)$

$$
\begin{array}{lrcl}
\text{Encrypt:} & h(n_2, k_2),\ k_3 & \rightarrow & \{k_3\}_{k_2} \\
\text{Unwrap:} & h(n_2, k_2),\ \{k_3\}_{k_2} & \xrightarrow{\text{new } n_3} & h(n_3, k_3) \\
\text{Set\_wrap:} & h(n_3, k_3) & \rightarrow & \text{wrap}(n_3) \\
\text{Wrap:} & h(n_3, k_3),\ h(n_1, k_1) & \rightarrow & \{k_1\}_{k_3} \\
\text{Intruder:} & \{k_1\}_{k_3},\ k_3 & \rightarrow & k_1
\end{array}
$$

Fix decrypt/wrap, encrypt/unwrap..

Intruder knows: $h(n_1, k_1)$, $h(n_2, k_2)$, k_3
State: $\text{sensitive}(n_1), \text{extract}(n_1), \text{extract}(n_2)$

$$
\begin{array}{lrcl}
\text{Set\_wrap:} & h(n_2, k_2) & \rightarrow & \text{wrap}(n_2) \\
\text{Wrap:} & h(n_2, k_2), h(n_2, k_2) & \rightarrow & \{k_2\}_{k_2} \\
\text{Set\_unwrap:} & h(n_2, k_2) & \rightarrow & \text{unwrap}(n_2) \\
\text{Unwrap:} & h(n_2, k_2), \{k_2\}_{k_2} & \xrightarrow{\text{new } n_4} & h(n_4, k_2) \\
\text{Wrap:} & h(n_2, k_2), h(n_1, k_1) & \rightarrow & \{k_1\}_{k_2} \\
\text{Set\_decrypt:} & h(n_4, k_2) & \rightarrow & \text{decrypt}(n_4) \\
\text{SDecrypt:} & h(n_2, k_2), \{k_1\}_{k_2} & \rightarrow & k_1
\end{array}
$$

More Attacks on PKCS#11

The "Unwrap to non-sensitive attack"

Suppose $h(n_1, k_1)$ is a handle to a sensitive key

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

Unwrap :

$$h(n_2, k_2), \{k_1\}_{k_2}, \mathcal{T};\ \text{unwrap}(n_2) \xrightarrow{\text{new } n_3} h(n_3, k_1);\ \text{extract}(n_3),\ T$$

Where $\mathcal{T}$ has sensitive=false

GetValue: $h(n_3, k_1) \quad \rightarrow \quad k_1$

And More Attacks on PKCS#11

The "wrap with non-sensitive" attack'

Suppose $h(n_1, k_1)$ handle to a sensitive key, $h(n_2, k_2)$ has sensitive=false

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

GetValue: $h(n_2, k_2) \rightarrow k_2$

Attacker decrypts $\{k_1\}_{k_2}$ himself to obtain k_1

Other PKCS#11 Problems

- Wrap with weaker key
- ECB split on DES keys
- Unwrap with PKCS#1.5 or CBC-PAD: error codes can lead to padding oracle attacks
  (we will look at this in more detail in the practical)

In the next half

- How to fix PKCS#11
- Trusted Keys
- Wrap formats
- Restricted Templates
- Programming PKCS#11 with Bees
- The Tookan analysis tool

Fixing PKCS#11

Many smartcard maufacturers remove all the wrapping and unwrapping functionality from PKCS#11

This might be ok for smartcards, but other devices with richer functionality (like HSMs) need to do these operations

There are in fact several ways to build secure interfaces using the standard, many with security proofs

Some are included in the standard v2.20, some are extensions (note that not many devices actually implement v2.20)

Unwrap Templates

Inside the template of a key k, we can give a pointer to another template which will be given to all keys unwrapped using k

This is useful: we can say that all keys will be sensitive, all keys will have wrap=false, etc.

We can even give an unwrap template inside the unwrap template...

Could in theory give us a template of unbounded size, but the token memory is typically small.

Introduced in PKCS#11 v2.20 but not yet widely supported

Wrap Format

We saw that problems were caused by unwrapping a key with a different template from its original one

We can prevent this by binding the attributes to the encrypted key

The Eracom HSM range already includes a method for this

Can we verify this if the attribute policy is good? Start by formalising attribute policy

$$\text{KeyGenerate}: \quad \xrightarrow{\text{new } n_1, k_1} \quad h(n_1, k_1);\ L(n_1), \neg\text{extract}(n_1)$$

Wrap :

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

Unwrap :

$$h(x_2, y_2), \{y_1\}_{y_2};\ \text{unwrap}(x_2) \quad \xrightarrow{\text{new } n_1} \quad h(n_1, y_1);\ L(n_1)$$

$$\begin{array}{lrcl} \text{Encrypt}: & h(x_1, y_1), y_2;\ \text{encrypt}(x_1) & \rightarrow & \{y_2\}_{y_1} \\ \text{Decrypt}: & h(x_1, y_1), \{y_2\}_{y_1};\ \text{decrypt}(x_1) & \rightarrow & y_2 \end{array}$$

$$\begin{array}{lrcl} \text{Set\_Encrypt}: & h(x_1, y_1);\ \neg\text{encrypt}(x_1) & \rightarrow & \text{encrypt}(x_1) \\ \text{UnSet\_Encrypt}: & h(x_1, y_1);\ \text{encrypt}(x_1) & \rightarrow & \neg\text{encrypt}(x_1) \\ \vdots & & \vdots & \end{array}$$

KeyGenerate : $\xrightarrow{\text{new } n_1,k_1} \quad h(n_1, k_1);\ A(n_1)$

Wrap :

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

Unwrap :

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

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

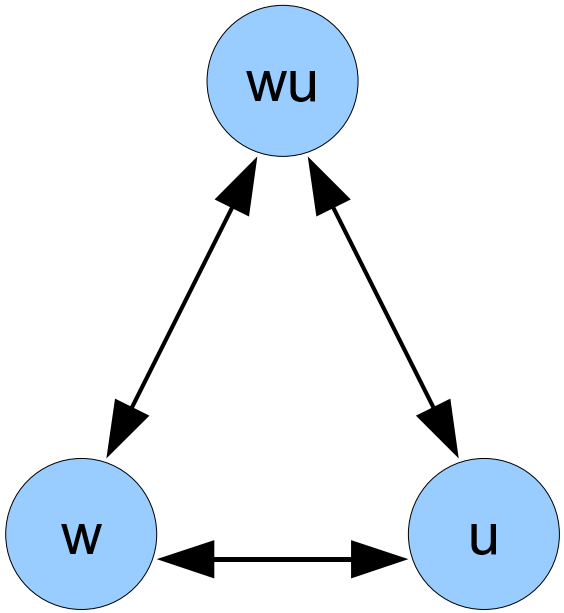
Decrypt : $h(x_1, y_1), \{y_2\}_{y_1};\ \text{decrypt}(x_1) \quad \rightarrow \quad y_2$

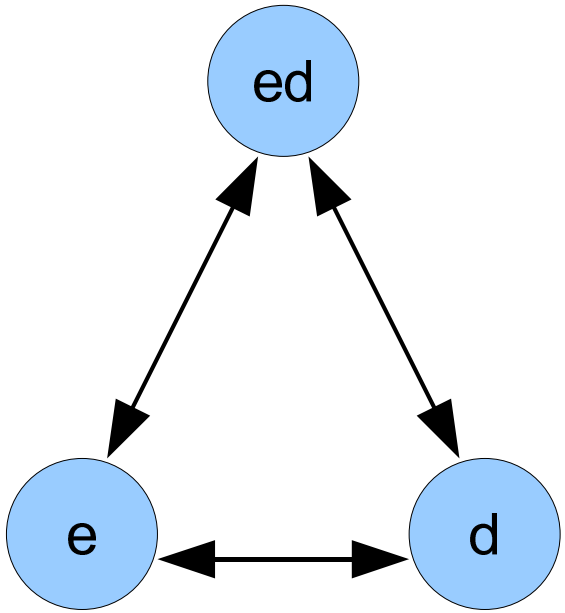
Set_Attribute_Value : $h(x_1, y_1);\ A_1(x_1) \quad \rightarrow \quad A_2(x_1)$

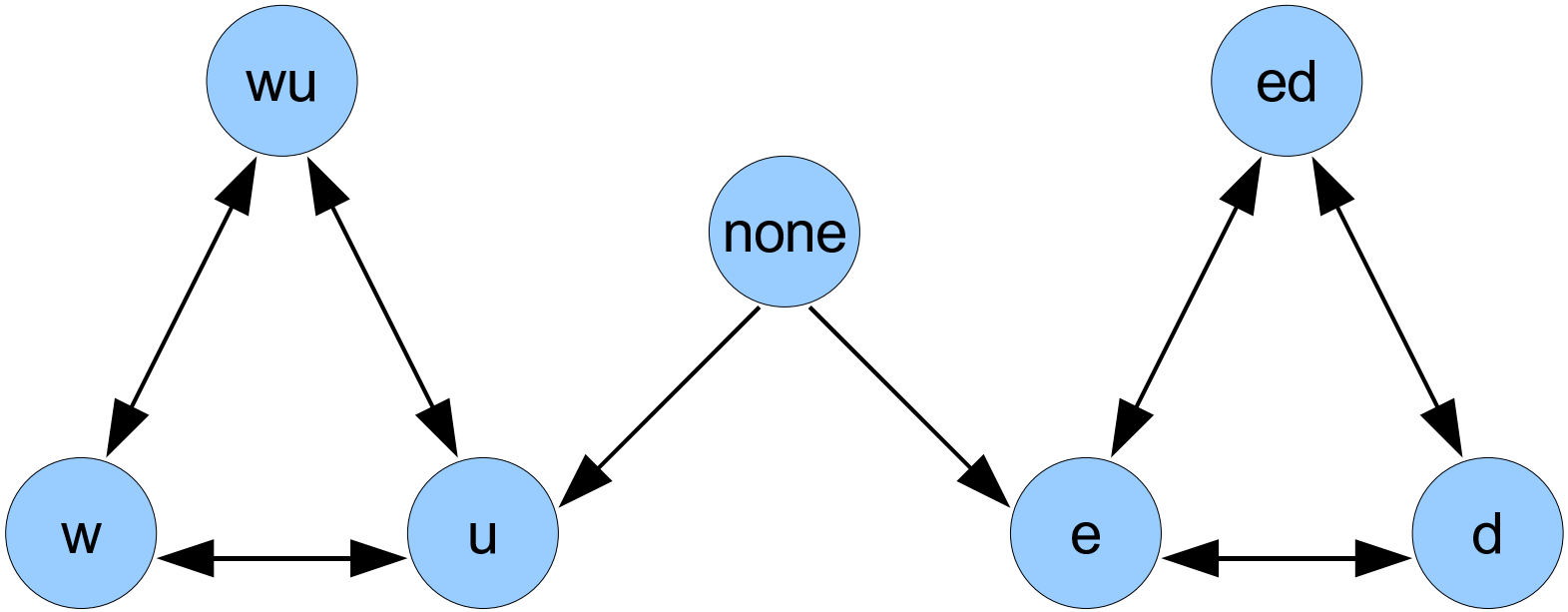
Attribute Policy

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.


wu
w
u


ed
e
d


wu
ed
none
w
u
e
d

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

- 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 $(\#\text{ep})^2$
  - for each key, get one handle for each endpoint

Intruder always starts with his own key
so require $\#\text{ep} + 1$ keys and $(\#\text{ep} + 1)^2$ handles

$$\text{KeyGenerate} : \xrightarrow{\text{new } n_1, k_1} h(n_1, k_1); A(n_1)$$

Wrap :

$$h(x_1, y_1), h(x_2, y_2);\ \text{wrap}(x_1), A(x_2) \xrightarrow{\text{new } m_k} \{y_2\}_{y_1}, \{m_k\}_{y_1}\ \text{hmac}_{m_k}(y_2, \mathcal{A})$$

Unwrap :

$$h(x_2, y_2), \{y_1\}_{y_2}, \{x_m\}_{y_2},\ \text{hmac}_{x_m}(y_1, \mathcal{A});\ \text{unwrap}(x_2) \xrightarrow{\text{new } n_1} h(n_1, y_1);\ A(n_1)$$

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

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

$P = (\{e, d, ed, w, u, wu\}, \rightarrow)$ (where $\rightarrow$ makes the obvious cliques)

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 $ed(n_1), wu(n_2), ed(n_i)$.

Trace:

Wrap: (ed)	$h(n_2, k_2)$, $h(n_i, k_i) \rightarrow$ $\{k_i\}_{k_2}$, $\{k_3\}_{k_2}$, $hmac_{k_3}(k_i, ed)$
Unwrap: (wu)	$h(n_2, k_2)$, $\{k_i\}_{k_2}$, $\{k_i\}_{k_2}$, $hmac_{k_i}(k_i, wu) \rightarrow h(n_2, k_i)$
Wrap: (ed)	$h(n_2, k_i)$, $h(n_1, k_1) \rightarrow$ $\{k_1\}_{k_i}$, $\{k_3\}_{k_i}$, $hmac_{k_3}(k_1, ed)$
Decrypt:	k_i, $\{k_1\}_{k_i} \rightarrow k_1$

Revised API

Wrap :

$$h(x_1, y_1), h(x_2, y_2);\ \mathsf{wrap}(x_1), A(x_2) \xrightarrow{\text{new } m_k} \{y_2\}_{y_1}, \{m_k\}_{y_1}\ \mathsf{hmac}_{m_k}(y_2, \mathcal{A}, y_1)$$

Unwrap :

$$h(x_2, y_2), \{y_1\}_{y_2}, \{x_m\}_{y_2},\ \mathsf{hmac}_{x_m}(y_1, \mathcal{A}, y_2);\ \mathsf{unwrap}(x_2) \xrightarrow{\text{new } n_1} h(n_1, y_1);\ A(n_1)$$

Property 2 now verified by SATMC

Can also verify attribute policy is enforced

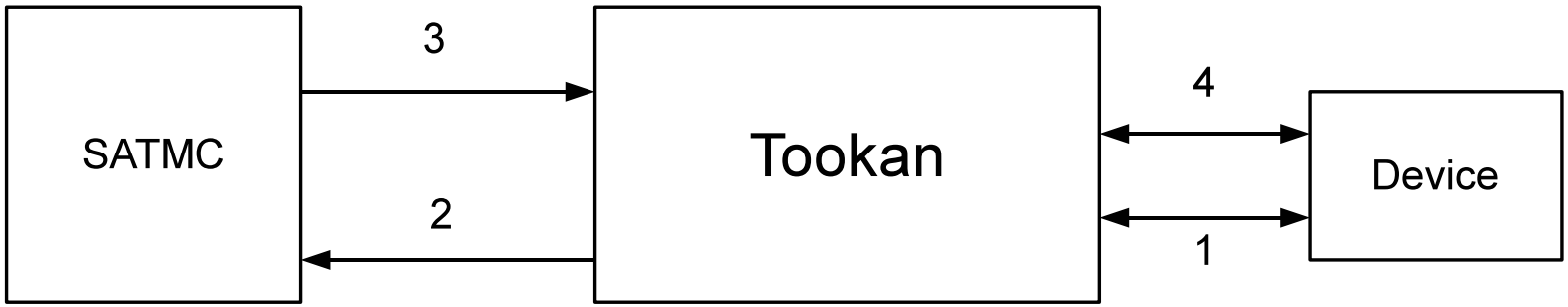
Other Fixes

Using no new cryptographic mechanisms, v2.11 of standard:
Allow only generate templates {wu,ed}, unwrap templates { eu },
all attributes sticky on and off.
Requires one key for each direction of communication.
See [Bortolozzo, Centenaro, Focardi, S. '10]

Keys can be marked wrap_with_trusted
A key marked wrap_with_trusted can only be wrapped with a key marked trusted
See [Delaune, Kremer, Steel '08], [Fröschle, FAST '11]

TOOKAN

'Tool for cryptoKi Analysis'




TOOKAN

Device		can wrap sensitive keys	Category of attacks found	
Brand	Model		wrap-decrypt variant	read sensitive/ unextractable
Aladdin	eToken PRO	✓	✓	
Athena	ASEKey			
Bull	Trustway RCI	✓	✓	
Eutron	Crypto Id. ITSEC			
Feitian	StorePass2000	✓	✓	✓
Feitian	ePass2000	✓	✓	✓
Feitian	ePass3003Auto	✓	✓	✓
Gemalto	SEG			
MXI	Stealth MXP Bio			
RSA	SecurID 800			✓
SafeNet	iKey 2032			
Sata	DKey	✓	✓	✓
ACS	ACOS5			
Athena	ASE Smartcard			
Gemalto	Cyberflex V2	✓	✓	
Gemalto	SafeSite V1			
Gemalto	SafeSite V2	✓	✓	✓
Siemens	CardOS			✓

Sample results on smartcards/USB tokens

The Bee Library

Programming PKCS#11 directly in C can be tedious and error prone: lots of housekeeping for pointers and memory, structures for templates, etc.

Many wrappers exist for programming in C or other languages: IAIK, PyKCS11, pkcs11-helper, Bees

Bees was developed in the Tookan project and includes a C++ and Java interface

We will examine java-Bee now via some examples

Example: Opening a session, get info

```
import bee.*;
public class GetInfo
{
   public static final String LIB =
      "/usr/local/lib/opencryptoki/libopencryptoki.so";
   public static final String PIN = "12345";
   public static void main(String[] args)
                                    throws BeeException
    {
        Bee b = new Bee(LIB, PIN, 0);
        TokenInfo info = b.getTokenInfo();
        System.out.println(info);
        b.logout();
    }
```

Example: Opening a session, get info

```
ObjectHandle[] objs = b.find(new Template());
System.out.println(objs.length + " object(s) found");
for (ObjectHandle o : objs)
{
 String label;
 try {label = o.getTemplate().getLabel();}
 catch (BeeException e) {label = e.getMessage();}
 System.out.println("\t" + i + ": " + label);
}
```

Example: Creating and Manipulating a Key

```
Template t = new Template();
t.setToken(true);
ObjectHandle key = b.generateKey(t);

t = new Template();
t.setEncrypt(true);
key.setTemplate(t); //calls SetAttributes
```

Example: Using a Key

```
byte[] val = b.symWrap(key1, key2);

byte[] val2 = b.symEncrypt(val2, key2);

Template bar = new Template();
bar.setObjClass(new Pkcs11Class(Pkcs11Class.SECRET_KEY));
bar.setKeyType(new KeyType(KeyType.AES));

val2 foo = b.symUnwrap(val, key1, bar);
```

(Catch BeeExceptions!)

Alternatives to PKCS#11

Cortier & Steel API

- First presented at ESORICS 2009
- Keys assigned fixed attributes at creation time: level and agent identifiers
- Extended to revocation (CCS '12) and asymmetric crypto (to appear)
- A version to be implemented in a French MoD project

Cachin & Chandran API

- Presented at CSF 2009
- Keys attributes evolve over time with usage (so central server required)
- Implemented in an IBM product

Alternatives to PKCS#11 - 2

Both proposals have some points in common:

- Attributes of key tied to key value on export
- Key role separation enforced
- Authenticated encryption schemes

Will new industry proposals, e.g. KMIP reflect this?

Summary

- RSA PKCS#11 is ubiquitous in key management APIs
- Many attacks, many approaches to securing
- Tookan: an automated audit tool
- Bees: a java library for PKCS#11 programming
- Alternatives emerging in academia and industry

Further Reading

RSA PKCS#11,
`www.rsa.com/rsalabs/node.asp?id=2133`

J. Clulow, *On The Security of PKCS#11*, CHES 2003

S. Delaune, S. Kremer and G. Steel, *Formal Analysis of PKCS#11 and Proprietary Extensions*, CSF 2008

M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, *Attacking and Fixing PKCS#11 Security Tokens*, CCS 2010

S. Fröschle and G. Steel, *Formal Analysis of PKCS#11 with Unbounded Fresh Data*, WITS 2009