A Formal Analysis of Authentication in the TPM

Stéphanie Delaune[1], Steve Kremer[1], Mark D. Ryan[2], and Graham Steel[1]

[1] LSV, ENS Cachan & CNRS & INRIA Saclay Île-de-France, France
[2] School of Computer Science, University of Birmingham, UK

Thursday, September 30th, 2010

TPM - What is it?

Trusted Platform Module

Hardware chip designed to enable commodity computers to achieve greater levels of security than is possible in software alone.

TPM - What is it?

Trusted Platform Module

Hardware chip designed to enable commodity computers to achieve greater levels of security than is possible in software alone.

- more than 200 millions currently in existence (mostly in laptops)
  $\longrightarrow$ already used by some applications (*e.g.* Disk encryption)
- specified by the Trusted Computing Group
  $\longrightarrow$ more than 700 pages of specification

`http://www.trustedcomputinggroup.org`

TPM functionality

Secure storage:

- TPM stores keys and other sensitive data in its shielded memory
- A user can store content that is encrypted by keys only available to the TPM.

TPM functionality

Secure storage:

- TPM stores keys and other sensitive data in its shielded memory
- A user can store content that is encrypted by keys only available to the TPM.

Platform authentication:

- Each TPM chip has a unique and secret key
- A platform can obtain keys by which it can authenticate itself reliably.

TPM functionality

Secure storage:

- TPM stores keys and other sensitive data in its shielded memory
- A user can store content that is encrypted by keys only available to the TPM.

Platform authentication:

- Each TPM chip has a unique and secret key
- A platform can obtain keys by which it can authenticate itself reliably.

TPM contains also some internal memory slots called PCRs.

Platform measurement and reporting: A platform can create reports of its integrity and configuration state that can be relied on by a remote verifier.
$\longrightarrow$ used to ensure that a PC is in a particular configuration before starting an application.

Our contributions

$\longrightarrow$ formalise some commands and analyse them using an automated tool.

Formalise commands and security properties ...

- we model a collection of 4 TPM commands
  $\longrightarrow$ *e.g.* CreateWrapKey, LoadKey2, ...
- we identify security properties
  $\longrightarrow$ injective agreement properties modelled as correspondence properties

Our contributions

⟶ formalise some commands and analyse them using an automated tool.

Formalise commands and security properties ...

- we model a collection of 4 TPM commands
  ⟶ *e.g.* CreateWrapKey, LoadKey2, . . .
- we identify security properties
  ⟶ injective agreement properties modelled as correspondence properties

... in a way suitable to allow an automated tool to perform the analysis.

Our contributions

$\longrightarrow$ formalise some commands and analyse them using an automated tool.

Formalise commands and security properties ...

- we model a collection of 4 TPM commands
  $\longrightarrow$ *e.g.* CreateWrapKey, LoadKey2, ...
- we identify security properties
  $\longrightarrow$ injective agreement properties modelled as correspondence properties

... in a way suitable to allow an automated tool to perform the analysis.

Analysis (with the ProVerif tool)

- we rediscover some known attacks and new variations of them
- we propose some fixes

Outline

1. Introduction
2. An Overview of the TPM
3. Modelling the TPM
4. Analysing the TPM with ProVerif
5. Conclusion

Outline

TPM key hierarchy

Cryptographic key

Keys are arranged in a tree structure and stored in the TPM memory
$\longrightarrow$ Storage Root Key created by a special command

Authdata

To each TPM object or resource (*e.g.* keys) is associated an authdata value

- A shared secret between the user process and the TPM
  $\longrightarrow$ a password that has to be cited to use the object or resource
- authdata is 20 bytes (160 bits)

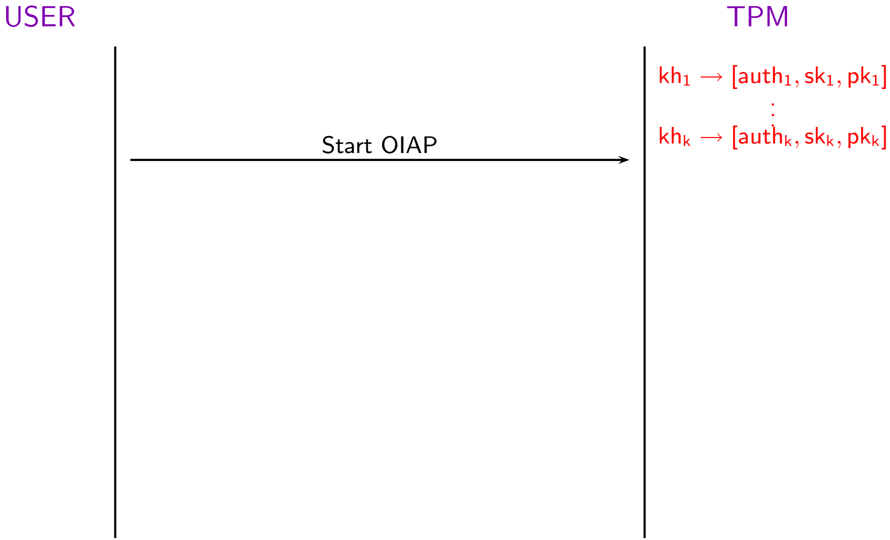
Authorisation Sessions

The TPM provides two kinds of authorisation sessions:

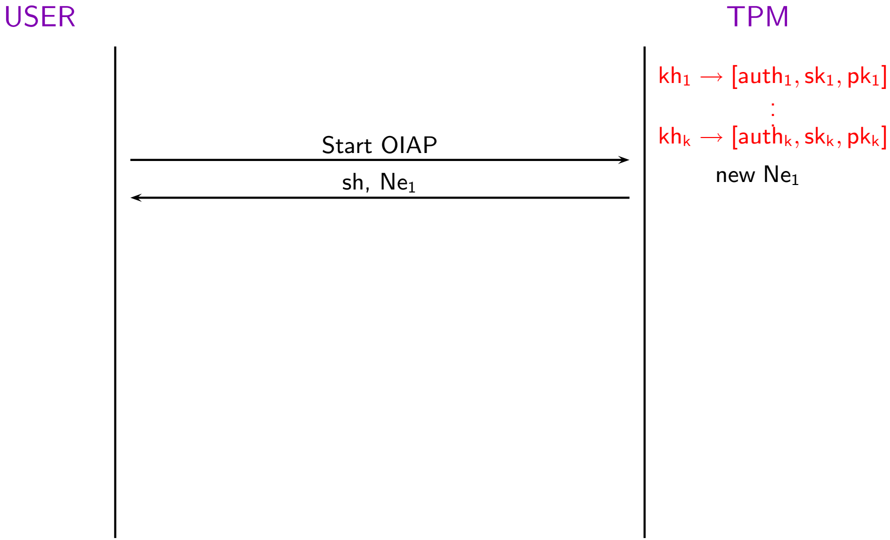
1. Object Independent Authorisation Protocol (OIAP)
   $\longrightarrow$ can manipulate any objects, but works only for certain commands

2. Object Specific Authorisation Protocol (OSAP)
   $\longrightarrow$ it manipulates a specific object specified when the session is set up

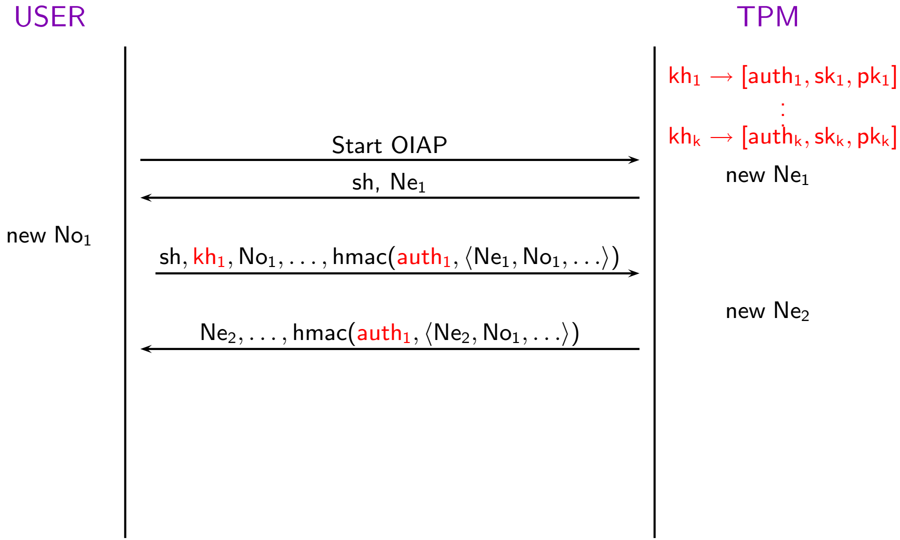
OIAP session



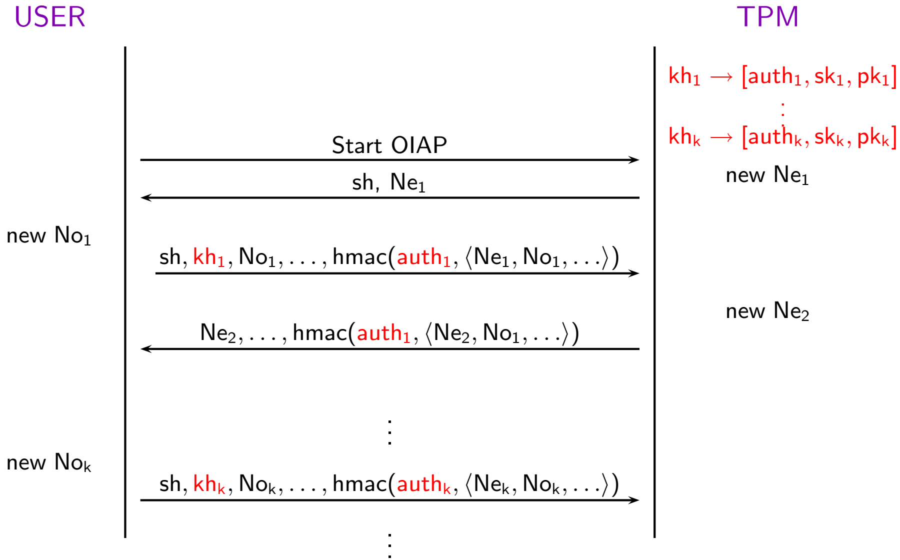
OIAP session



OIAP session



OIAP session



OSAP session

USER

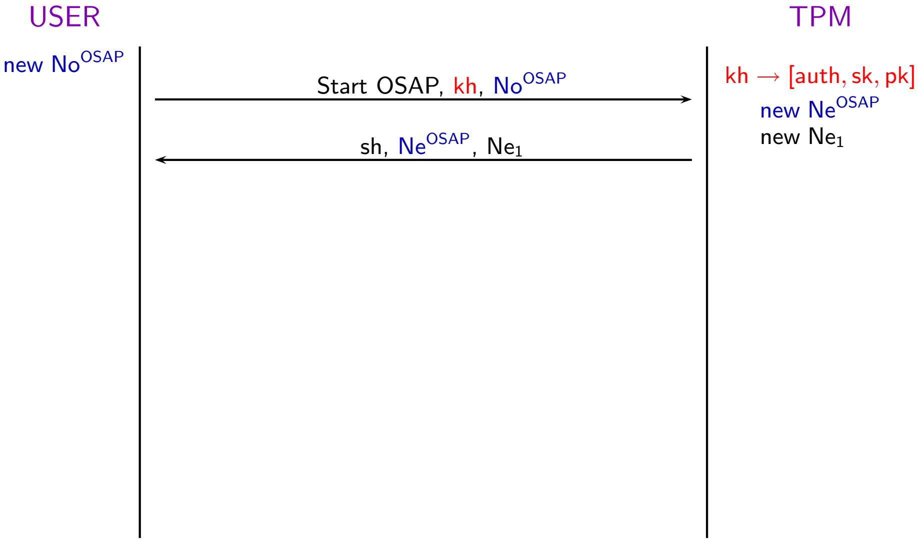
TPM

new $\mathsf{No}^{\mathsf{OSAP}}$

Start OSAP, kh, $\mathsf{No}^{\mathsf{OSAP}}$ →

$\mathsf{kh} \rightarrow [\mathsf{auth}, \mathsf{sk}, \mathsf{pk}]$

OSAP session



OSAP session

USER | TPM

new No^{OSAP}

Start OSAP, kh, No^{OSAP} →

$kh \rightarrow [auth, sk, pk]$

new Ne^{OSAP}

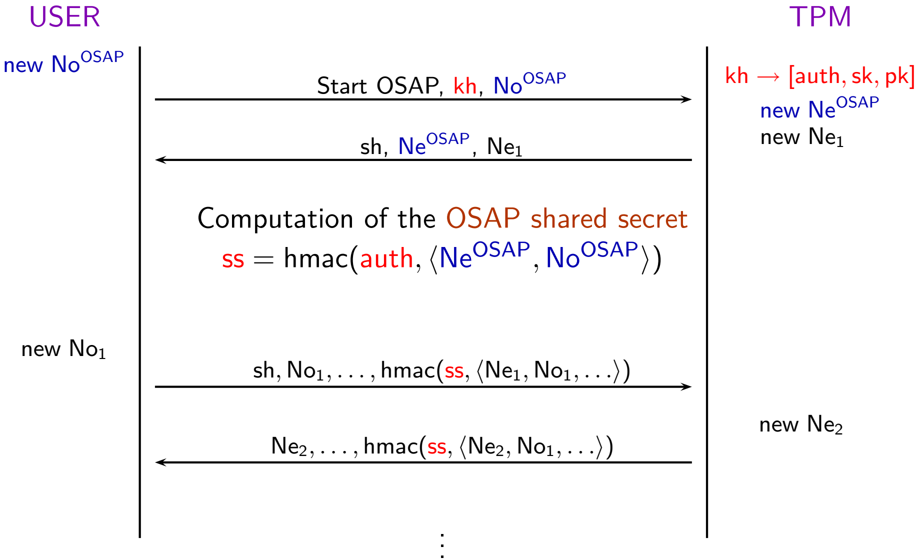
new Ne_1

← sh, Ne^{OSAP}, Ne_1

Computation of the OSAP shared secret

$$ss = hmac(auth, \langle Ne^{OSAP}, No^{OSAP} \rangle)$$

OSAP session



TPM-CreateWrapKey

Goal: Create the key and returns it protected by an encryption.

TPM-CreateWrapKey

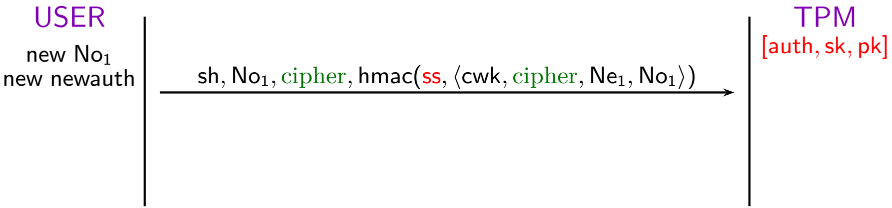
Goal: Create the key and returns it protected by an encryption.

Description: Assuming an OSAP session has been established

TPM-CreateWrapKey

Goal: Create the key and returns it protected by an encryption.

Description: Assuming an OSAP session has been established



where:

- $\mathrm{cipher} = \mathsf{senc}(\mathsf{newauth}, \mathsf{hash}(\mathsf{ss}, \mathsf{Ne}_1))$

TPM-CreateWrapKey

Goal: Create the key and returns it protected by an encryption.

Description: Assuming an OSAP session has been established

USER		TPM [auth, sk, pk]
new No_1 new newauth	$\xrightarrow{\mathsf{sh}, \mathsf{No}_1, \mathrm{cipher}, \mathsf{hmac}(\mathsf{ss}, \langle \mathsf{cwk}, \mathrm{cipher}, \mathsf{Ne}_1, \mathsf{No}_1 \rangle)}$	
	$\xleftarrow{\mathsf{Ne}_2, \mathsf{pk}(\mathsf{SK}), \mathrm{wrp}, \mathsf{hmac}(\mathsf{ss}, \langle \mathsf{cwk}, \mathrm{wrp}, \mathsf{pk}(\mathsf{SK}), \mathsf{Ne}_2, \mathsf{No}_1 \rangle)}$	new Ne_2 new SK

where:

- $\mathrm{cipher} = \mathsf{senc}(\mathsf{newauth}, \mathsf{hash}(\mathsf{ss}, \mathsf{Ne}_1))$
- $\mathrm{wrp} = \mathsf{wrap}(\langle \mathsf{SK}, \mathsf{newauth}, \mathsf{tpmproof} \rangle, \mathsf{pk})$

TPM-CertifyKey

Goal: allow a user to obtain a certificate on a key.

TPM-CertifyKey

Goal: allow a user to obtain a certificate on a key.

Description: Assuming two OAIP sessions have been established

USER		TPM
new No_1		$kh_1 \rightarrow [auth_1, sk_1, pk_1]$
new No_2		$kh_2 \rightarrow [auth_2, sk_2, pk_2]$
new N	$N, sh_1, kh_1, No_1, hmac(auth_1, \langle cfk, N, Ne_1, No_1\rangle)$ $sh_2, kh_2, No_2, hmac(auth_2, \langle cfk, N, Ne_2, No_2\rangle)$ $\longrightarrow$	
		new Ne'_1 new Ne'_2
	$certif,\ Ne'_1, hmac(auth_1, \langle cfk, N, Ne'_1, No_1, certif\rangle)$ $Ne'_2, hmac(auth_2, \langle cfk, N, Ne'_2, No_2, certif\rangle)$ $\longleftarrow$	

where:

- $certif = cert(pk_2, sk_1)$

Outline

The ProVerif tool (B. Blanchet)

Available on line:

`http://www.proverif.ens.fr/`

Input: processes written in applied pi calculus

Characteristics

- unbounded number of sessions
- primitives given by an equational theory
- security properties: (strong) secrecy, correspondence properties, equivalence properties
- sound but not complete
  $\longrightarrow$ sometimes, the tool reports some false attacks

Modelling

How to get rid of non-monotonic global state?

- only one command is executed in each OIAP or OSAP session
  $\longrightarrow$ the TPM imposes this restriction itself for certain command (*e.g.* CreateWrapKey)
  $\longrightarrow$ some tools that provides software-level API's also implement it
- do not allow keys to be deleted from the memory of the TPM
  $\longrightarrow$ we allow an unbounded number of keys to be loaded

Modelling

How to get rid of non-monotonic global state?

- only one command is executed in each OIAP or OSAP session
  $\longrightarrow$ the TPM imposes this restriction itself for certain command (*e.g.* CreateWrapKey)
  $\longrightarrow$ some tools that provides software-level API's also implement it
- do not allow keys to be deleted from the memory of the TPM
  $\longrightarrow$ we allow an unbounded number of keys to be loaded

Modelling the key table

- an entry
- private functions to model a lookup in the table

Modelling

How to get rid of non-monotonic global state?

- only one command is executed in each OIAP or OSAP session
  $\longrightarrow$ the TPM imposes this restriction itself for certain command (*e.g.* CreateWrapKey)
  $\longrightarrow$ some tools that provides software-level API's also implement it
- do not allow keys to be deleted from the memory of the TPM
  $\longrightarrow$ we allow an unbounded number of keys to be loaded

Modelling the key table

- an entry $\mathsf{handle}(\mathsf{auth}, \mathsf{sk})$
- private functions to model a lookup in the table

$$\mathsf{getAuth}(\mathsf{handle}(\mathsf{auth}, \mathsf{sk})) = \mathsf{auth}$$
$$\mathsf{getSK}(\mathsf{handle}(\mathsf{auth}, \mathsf{sk})) = \mathsf{sk}$$

Modelling

How to get rid of non-monotonic global state?

- only one command is executed in each OIAP or OSAP session
  $\longrightarrow$ the TPM imposes this restriction itself for certain command (*e.g.* CreateWrapKey)
  $\longrightarrow$ some tools that provides software-level API's also implement it
- do not allow keys to be deleted from the memory of the TPM
  $\longrightarrow$ we allow an unbounded number of keys to be loaded

Modelling the key table

- an entry $\mathsf{handle}(\mathsf{auth}, \mathsf{sk})$
- private functions to model a lookup in the table

$$\mathsf{getAuth}(\mathsf{handle}(\mathsf{auth}, \mathsf{sk})) = \mathsf{auth}$$
$$\mathsf{getSK}(\mathsf{handle}(\mathsf{auth}, \mathsf{sk})) = \mathsf{sk}$$

$\longrightarrow$ false attacks based on the hypothesis that the attacker knows $\mathsf{handle}(\mathsf{auth}_1, \mathsf{sk})$ and $\mathsf{handle}(\mathsf{auth}_2, \mathsf{sk})$.

Modelling

How to get rid of non-monotonic global state?

- only one command is executed in each OIAP or OSAP session
  $\longrightarrow$ the TPM imposes this restriction itself for certain command (*e.g.* CreateWrapKey)
  $\longrightarrow$ some tools that provides software-level API's also implement it
- do not allow keys to be deleted from the memory of the TPM
  $\longrightarrow$ we allow an unbounded number of keys to be loaded

Modelling the key table

- an entry $\mathsf{handle}(\mathsf{auth}, \mathsf{seed})$
- private functions to model a lookup in the table

$$\mathsf{getAuth}(\mathsf{handle}(\mathsf{auth}, \mathsf{seed})) = \mathsf{auth}$$
$$\mathsf{getSK}(\mathsf{handle}(\mathsf{auth}, \mathsf{seed})) = \mathsf{hsk}(\mathsf{auth}, \mathsf{seed})$$

Modelling Commands

Two processes for each command:

- a USER process models a honest user who makes a call to the TPM
- a TPM process models the TPM itself

$\longrightarrow$ the attacker schedules honest user actions

Security Properties

- secrecy of the private keys stored in the device (if their parent key is not compromised).

Security Properties

- secrecy of the private keys stored in the device (if their parent key is not compromised).

[TPM specification Part I, p. 60]

> *« The design criterion of the protocols is to allow for ownership authentication, command and parameter authentication and prevent replay and man in the middle attacks. »*

Security Properties

- secrecy of the private keys stored in the device (if their parent key is not compromised).

[TPM specification Part I, p. 60]

> « *The design criterion of the protocols is to allow for ownership authentication, command and parameter authentication and prevent replay and man in the middle attacks.* »

Our interpretation:

- authentication of user commands
  $\longrightarrow$ intuitively ensured by the authorisation hmacs
- authentication of the TPM
  $\longrightarrow$ intuitively ensured by the hmacs returned by the TPM

$\longrightarrow$ We formalise these as correspondance properties

Outline

Methodology

We consider four commands:

CreateWrapKey, LoadKey, CertifyKey, and UnBind.

Step 1: each command in isolation

- Configuration 1: two honest keys $[\mathsf{auth}_1, \mathsf{sk}_1, \mathsf{pk}_1]$, $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$
- Configuration 2: $+$ an additional honest key $[\mathsf{auth}_2, \mathsf{sk}'_2, \mathsf{pk}'_2]$
- Configuration 3: $+$ a dishonest key $[\mathsf{auth}_i, \mathsf{sk}_i, \mathsf{pk}_i]$

Methodology

We consider four commands:

CreateWrapKey, LoadKey, CertifyKey, and UnBind.

Step 1: each command in isolation

- Configuration 1: two honest keys $[\mathsf{auth}_1, \mathsf{sk}_1, \mathsf{pk}_1]$, $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$
- Configuration 2: + an additional honest key $[\mathsf{auth}_2, \mathsf{sk}'_2, \mathsf{pk}'_2]$
- Configuration 3: + a dishonest key $[\mathsf{auth}_i, \mathsf{sk}_i, \mathsf{pk}_i]$

$\longrightarrow$ We propose a fix version of each of this command

Step 2: the four commands together

- Configuration 4: an honest key $[\mathsf{auth}, \mathsf{sk}, \mathsf{pk}]$
  + a dishonest key $[\mathsf{auth}_i, \mathsf{sk}_i, \mathsf{pk}_i]$

CertifyKey command (1)

Configuration 1: $[\mathsf{auth}_1, \mathsf{sk}_1, \mathsf{pk}_1]$, $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$.

Attack: swap the two authorisation hmacs.
$\longrightarrow$ TPM sends $\mathsf{cert}(\mathsf{pk}_1, \mathsf{sk}_2)$ whereas the user asked for $\mathsf{cert}(\mathsf{pk}_2, \mathsf{sk}_1)$

Why is this possible ?

$\longrightarrow$ the two authorisation hmacs look very similar

$$\mathsf{hmac}(\mathsf{auth}_1, \langle \mathsf{cfk}, \mathsf{N}, \mathsf{Ne}_1, \mathsf{No}_1 \rangle)$$
$$\mathsf{hmac}(\mathsf{auth}_2, \langle \mathsf{cfk}, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle)$$

CertifyKey command (1)

Configuration 1: $[\mathsf{auth}_1, \mathsf{sk}_1, \mathsf{pk}_1]$, $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$.

Attack: swap the two authorisation hmacs.
$\longrightarrow$ TPM sends $\mathsf{cert}(\mathsf{pk}_1, \mathsf{sk}_2)$ whereas the user asked for $\mathsf{cert}(\mathsf{pk}_2, \mathsf{sk}_1)$

Why is this possible ?

$\longrightarrow$ the two authorisation hmacs look very similar

$$\mathsf{hmac}(\mathsf{auth}_1, \langle \mathsf{cfk}, \mathsf{N}, \mathsf{Ne}_1, \mathsf{No}_1 \rangle)$$
$$\mathsf{hmac}(\mathsf{auth}_2, \langle \mathsf{cfk}, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle)$$

A possible fix:

$$\mathsf{hmac}(\mathsf{auth}_1, \langle \mathsf{cfk}_1, \mathsf{N}, \mathsf{Ne}_1, \mathsf{No}_1 \rangle)$$
$$\mathsf{hmac}(\mathsf{auth}_2, \langle \mathsf{cfk}_2, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle)$$

$\longrightarrow$ the two correspondence properties hold on Configuration 1

CertifyKey command (2)

Configuration 2: $[\mathsf{auth}_1, \mathsf{sk}_1, \mathsf{pk}_1]$, $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$, $[\mathsf{auth}_2, \mathsf{sk}_2', \mathsf{pk}_2']$

Attack: exchange the key handle $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$ with $[\mathsf{auth}_2, \mathsf{sk}_2', \mathsf{pk}_2']$.
$\longrightarrow$ TPM sends $\mathsf{cert}(\mathsf{pk}_2', \mathsf{sk}_1)$ whereas the user asked for $\mathsf{cert}(\mathsf{pk}_2, \mathsf{sk}_1)$.

Why is this possible?

$\longrightarrow$ the authorisation hmacs do not depend on the key but only on the authdata.

CertifyKey command (2)

Configuration 2: $[\mathsf{auth}_1, \mathsf{sk}_1, \mathsf{pk}_1]$, $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$, $[\mathsf{auth}_2, \mathsf{sk}_2', \mathsf{pk}_2']$

Attack: exchange the key handle $[\mathsf{auth}_2, \mathsf{sk}_2, \mathsf{pk}_2]$ with $[\mathsf{auth}_2, \mathsf{sk}_2', \mathsf{pk}_2']$.
$\longrightarrow$ TPM sends $\mathsf{cert}(\mathsf{pk}_2', \mathsf{sk}_1)$ whereas the user asked for $\mathsf{cert}(\mathsf{pk}_2, \mathsf{sk}_1)$.

Why is this possible?

$\longrightarrow$ the authorisation hmacs do not depend on the key but only on the authdata.

A possible fix: add the (digest of the) public part of the key.

$$\mathsf{hmac}(\mathsf{auth}_1, \langle \mathsf{cfk}_1, \mathsf{pk}_1, \mathsf{N}, \mathsf{Ne}_1, \mathsf{No}_1 \rangle)$$
$$\mathsf{hmac}(\mathsf{auth}_2, \langle \mathsf{cfk}_2, \mathsf{pk}_2, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle)$$

$\longrightarrow$ the two correspondence properties now hold on Configuration 2

CerifyKey command (3)

Configuration 3: as before + $[auth_i, sk_i, pk_i]$

Attack: replace the key to be certified by pk_i.
$\longrightarrow$ TPM sends $cert(pk_i, sk_1)$ whereas the user asked for $cert(pk_2, sk_1)$

How is this possible?
$\longrightarrow$ The two authorisation hmacs are linked together only through the nonce N (known by the attacker).

$$\begin{array}{ll} \text{Intercept} & hmac(auth_2, \langle cfk_2, pk_2, N, Ne_2, No_2\rangle) \\ \text{Send} & hmac(auth_i, \langle cfk_2, pk_i, N, Ne_2, No_2\rangle). \end{array}$$

CerifyKey command (3)

Configuration 3: as before + $[\mathsf{auth}_i, \mathsf{sk}_i, \mathsf{pk}_i]$

Attack: replace the key to be certified by pk_i.
$\longrightarrow$ TPM sends $\mathsf{cert}(\mathsf{pk}_i, \mathsf{sk}_1)$ whereas the user asked for $\mathsf{cert}(\mathsf{pk}_2, \mathsf{sk}_1)$

How is this possible?
$\longrightarrow$ The two authorisation hmacs are linked together only through the nonce N (known by the attacker).

$$\begin{array}{ll} \text{Intercept} & \mathsf{hmac}(\mathsf{auth}_2, \langle \mathsf{cfk}_2, \mathsf{pk}_2, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle) \\ \text{Send} & \mathsf{hmac}(\mathsf{auth}_i, \langle \mathsf{cfk}_2, \mathsf{pk}_i, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle). \end{array}$$

A possible fix:

$$\begin{array}{l} \mathsf{hmac}(\mathsf{auth}_1, \langle \mathsf{cfk}_1, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{N}, \mathsf{Ne}_1, \mathsf{No}_1 \rangle) \\ \mathsf{hmac}(\mathsf{auth}_2, \langle \mathsf{cfk}_2, \mathsf{pk}_2, \mathsf{pk}_1, \mathsf{N}, \mathsf{Ne}_2, \mathsf{No}_2 \rangle) \end{array}$$

$\longrightarrow$ the two correspondence properties now hold on Configuration 3

The 4 commands together

Configuration 4: an honest key $[\mathsf{auth}, \mathsf{sk}, \mathsf{pk}]$
+ a dishonest key $[\mathsf{auth}_i, \mathsf{sk}_i, \mathsf{pk}_i]$

$\longrightarrow$ we do not need to add more since the user can now create his own key and the attacker also.

Results:

ProVerif establishes the 8 correspondences properties. It fails to prove the injective version of one of them (false attack).

The 4 commands together

Configuration 4: an honest key [auth, sk, pk]
+ a dishonest key $[\mathsf{auth}_i, \mathsf{sk}_i, \mathsf{pk}_i]$

$\longrightarrow$ we do not need to add more since the user can now create his own key and the attacker also.

Results:

ProVerif establishes the 8 correspondences properties. It fails to prove the injective version of one of them (false attack).

All the files for our experiments are available on line at:

`http://www.lsv.ens-cachan.fr/~delaune/TPM/.`

Outline

Conclusion

Conclusion

- We formalise 4 commands of the TPM and their security properties
  $\longrightarrow$ injective agreement properties as correspondence properties
- Analysis with the ProVerif tool
  $\longrightarrow$ we rediscovered some attacks
  $\longrightarrow$ we propose some fixes

We foresee extending our model to deal with:

- Key migration commands;
- Platform Configuration Registers (PCRs);
- ...