Trusted Computing

System Security (CM0625, CM0631) 2023-24
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Introduction

Complex software systems are (eventually) **flawed**

Design flaws: **hard** to provide the intended security guarantees

Implementation flaws: even when design is correct, **bugs** might introduce vulnerabilities

Introduction

Formal models of security

Can we mathematically ***prove*** security?

Formal models of computer security can be used to “prove” that:

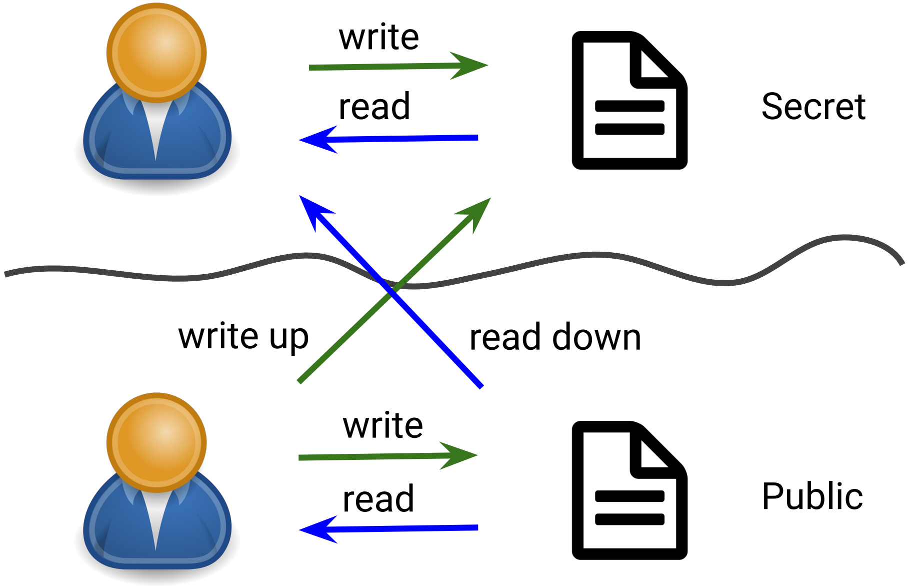
- **design** satisfies a set of security requirements
- **implementation** conforms to the design

Example: Bell - La Padula (BLP)

Definition: Information should never flow from a level to lower ones

- **Simple security**: Subjects cannot read from objects at a higher level
- ***-property**: Subjects cannot write into objects classified at a lower level

(plus **standard DAC**)

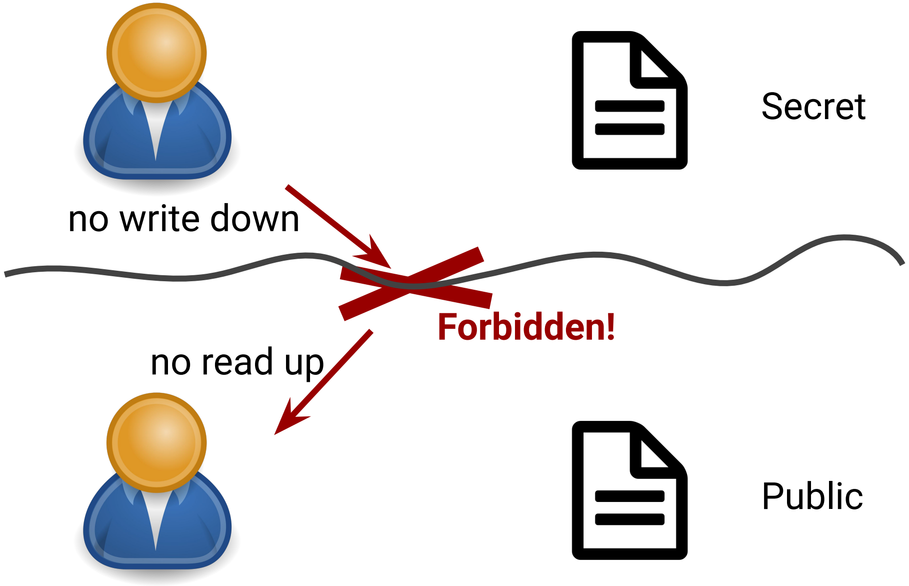


Example: Bell - La Padula (BLP)

Definition: Information should never flow from a level to lower ones

- **Simple security**: Subjects cannot read from objects at a higher level
- ***-property**: Subjects cannot write into objects classified at a lower level

(plus **standard DAC**)

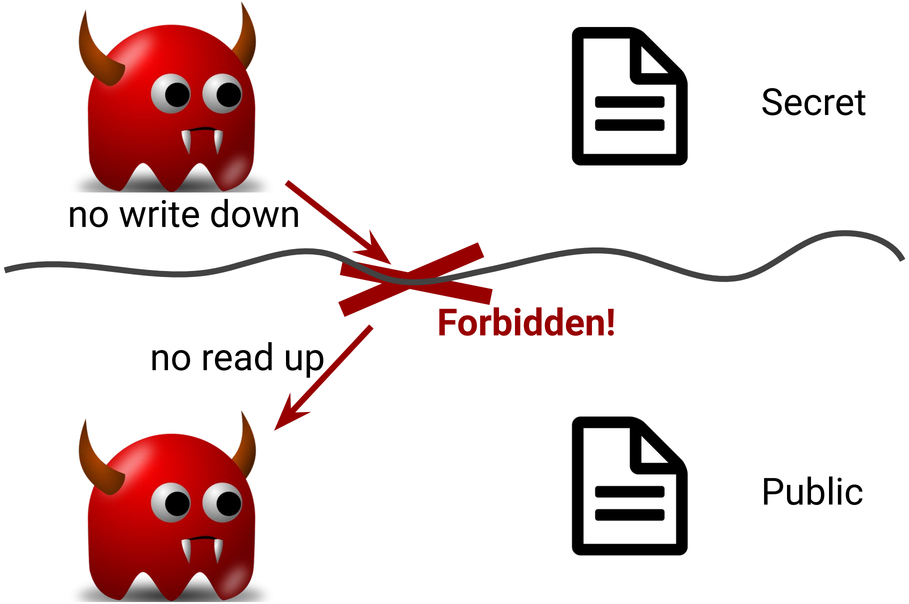


Example: Bell - La Padula (BLP)

Definition: Information should never flow from a level to lower ones

- **Simple security**: Subjects cannot read from objects at a higher level
- ***-property**: Subjects cannot write into objects classified at a lower level

(plus **standard DAC**)



BLP model

BLP can be stated **formally**

Assume: $S_1, \dots, S_m$ subjects, $O_1, \dots, O_n$ objects, $A_1, \dots, A_w$ access modes (e.g., **read**, **write**, **append**, …)

State: 3-tuple (**b**, **M**, **f**), defined as

b : **current access set** of triples (S_i, O_j, A_x) representing subject S_i accessing object O_j in mode A_x

M : **access matrix** of permitted access modes. M_{ij} contains modes for subject S_i accessing object O_j

f : **level function** assigning a security level to subjects and objects

$f_o(O_j)$ is the security level of object O_j

$f_s(S_i)$ is the security level of subject S_i

BLP model

Simple security: every triple of the form (S_i, O_j, **read**) in the current access set **b** has the property

$$f_s(S_i) \geq f_o(O_j)$$

***-property**: every triple of the form (S_i, O_j, **write**) in the current access set **b** has the property

$$f_s(S_i) \leq f_o(O_j)$$

In addition to **MAC**, BLP also enforces **DAC**, in terms of the access control matrix **M**. DAC is formalized as follows:

ds-property: if (S_i, O_j, A_x) is a current access in **b**, then access mode A_x is present in M_{ij}. That is

$$(S_i, O_j, A_x) \in b \Rightarrow A_x \in M_{ij}$$

BLP abstract operations

Get access: initiate access to object, i.e., adds (s,o,a) to **b**

Release access: release access to object, i.e., removes (s,o,a) from **b**

Change object level: change the value of $\mathbf{f_o(O_j)}$ for some object $\mathbf{O_j}$

Change current level: Change the value of $\mathbf{f_s(S_i)}$ for some subject $\mathbf{S_i}$

Give access permission: grant an access mode, i.e., add $\mathbf{A_x}$ to $\mathbf{M_{ij}}$

Revoke access permission: delete an access mode, i.e., remove $\mathbf{A_x}$ from $\mathbf{M_{ij}}$

Create an object: add a new object $\mathbf{O_j}$ with security level $\mathbf{f_o(O_j)}$

Delete an object: remove object $\mathbf{O_j}$

BLP security definition

Secure state: state (**b**, **M**, **f**) is secure if and only if every element of **b** satisfies the three properties

State transition: state (**b**, **M**, **f**) is changed by any operation that changes **b**, **M** or **f**

Secure system: a system starting from a **secure state** is **secure** iff any operation preserves the three properties

It is **theoretically possible** to prove that an actual system (or system design) is **secure** by proving that any action that affects the state of the system satisfies the three properties

For a complex system, such a proof can **hardly cover all cases**

⇒ Still, formal proof can lead to **more secure** design and implementation

Security of abstract operations

Get access: adds (S_i, O_j, **read**) to **b**

$$f_s(S_i) \geq f_o(O_j) \text{ and } \mathbf{read} \in M_{ij}$$

Get access: adds (S_i, O_j, **write**) to **b**

$$f_s(S_i) \leq f_o(O_j) \text{ and } \mathbf{write} \in M_{ij}$$

Change object/current level: change the value of $f_o(O_j)$ or $f_s(S_i)$

$$(S_i, O_j, \mathbf{read}) \in b \Rightarrow f_s(S_i) \geq f_o(O_j)$$
$$(S_i, O_j, \mathbf{write}) \in b \Rightarrow f_s(S_i) \leq f_o(O_j)$$

Revoke access permission: remove A_x from M_{ij}

$$(S_i, O_j, A_x) \notin b$$

When action violates the condition

- action is **forbidden** (error), or
- state should be updated, e.g., **release** accesses that violate the new permissions or levels (make the state secure)

Applications of BLP model

Implementing BLP in RBAC (1)

Constraint on users: For each subject **s** a security **clearance** L(**s**) is assigned

Permissions: For each role **r** and object **o**, assign **read/write** permission (access matrix)

Constraint on objects: For each object **o** a security **classification** L(**o**) is assigned

The read-level of a role **r,** denoted r-level(**r**), is the **least upper bound** of the security levels of the objects for which **read** is in the permissions of **r**

The write-level of a role **r**, denoted w-level(**r**), is the **greatest lower bound** of the security levels of the objects for which **write** is in the permissions of **r**

Implementing BLP in RBAC (2)

Constraint on role assignment: the clearance of the subject must **dominate** the r-level of the role and **be dominated** by the w-level of the role

$$L(S) \geq r\text{-}level(r)$$

$$L(S) \leq w\text{-}level(r)$$

The r-level of the role indicates the **least security classification** that dominates the level of objects readable from the role

Simple security property demands that a subject is assigned to a role only if the subject's clearance is **at least as high** as the r-level of the role

(dually for **write** access, *-property)

Implementing BLP in databases

Granularity of classification

- Entire database
- Individual tables (relations)
- Individual columns (fields)
- Individual rows (records)
- Individual elements

Granularity **affects** access control enforcement

Read access (simple security): For **entire databases** it is enough to allow access only when the subject clearance dominates database classification

Similarly, for individual tables, it is enough to only **allow queries** on tables whose classification is dominated by the subject clearance

Read access: individual columns

Example: Salary is **secret**

A user with clearance **public** executes query:

```
SELECT Name
    FROM Employee
    WHERE Salary > 50K
```

Name is **public** but query reveals information about **secret** salary!

⇒ forbidden (based on secret fields)

Name	**Salary**	**Phone**	**DID**
Alice	70K	041-2347...	2
Bob	50K	041-2348...	2
Carol	60K	041-2349...	1

Read access: individual rows

Example: rows with salary > 50K are **secret**

A user with clearance **public** executes query:

```
SELECT Name
    FROM Employee
```

If names of employees are known she deduces who has salary > 50K

⇒ what to do? (hard to fix)

Name	**Salary**	**Phone**	**DID**
Alice	70K	041-2347...	2
Bob	50K	041-2348...	2
Carol	60K	041-2349...	1

Polyinstantiation

Idea: add extra **public** rows with "fake" values

A user with clearance **public** executes query:

```
SELECT Name
   FROM Employee
```

Gets the **public** (fake) values and cannot deduce who has salary > 50K

Name	Salary	Phone	DID
Alice	70K	041-2347...	2
Alice	45K	041-2347...	2
Bob	50K	041-2348...	2
Carol	60K	041-2349...	1
Carol	48K	041-2349...	1

Trusted systems

Trust: confidence that system meets specifications, e.g., through **formal analysis** or **code review**

Trusted computing base (TCB): part of the system **enforcing** a particular policy, small enough to be **analyzed**

Evaluation: assessing if system has the **claimed security properties**

Trusted Platform Module (TPM)

TPM is a **hardware module** that is at the heart of a hardware/software approach to trusted computing

Standardized by the Trusted Computing Group

TPM is **integrated** in the CPU, the motherboard, or in smarcards

It is a hardware, **tamper resistant** Trusted Computing Base (TCB)

The TPM works with **TC-enabled software**, including the OS and applications

The software can be assured that the data it receives are **trustworthy**, and the system can be assured that the software itself is **trustworthy**

Three basic services: authenticated boot, certification, and encryption

Authenticated boot service

Responsible for booting the entire operating system, **assuring** that it is an **approved version** for use

Boot happens in **stages**:

- **Boot ROM** is loaded
- **Boot Block** on storage is loaded
- **Larger blocks** are brought in, until the full OS is loaded

At each stage, the TPM checks that **valid software** has been brought in, e.g. verifying a **digital signature** associated with the software

The TPM keeps a **tamper-evident log** of the loading process

⇒ a **cryptographic hash function** is used to detect any tampering with the log

Authenticated boot service

The tamper-resistant log contains a record that establishes exactly, **which version of the OS** and which of its **modules** are running

Trust boundary can be expanded to include additional hardware and application and utility software

⇒ **approved list** of hardware and software components

The TC-enabled system checks whether any new component

- is on the **approved list**
- is **digitally signed**
- has a serial number that has **not been revoked**

⇒ hardware, system software, and applications in a **well-defined state** with **approved components**.

Certification service

A mechanism to certify the (trusted) configuration to **other parties**

The TPM produces a **digital certificate** by **signing** a description of the configuration information using the TPM's private key

Other local or remote parties have **confidence** that an unaltered configuration is in use

Notice that:

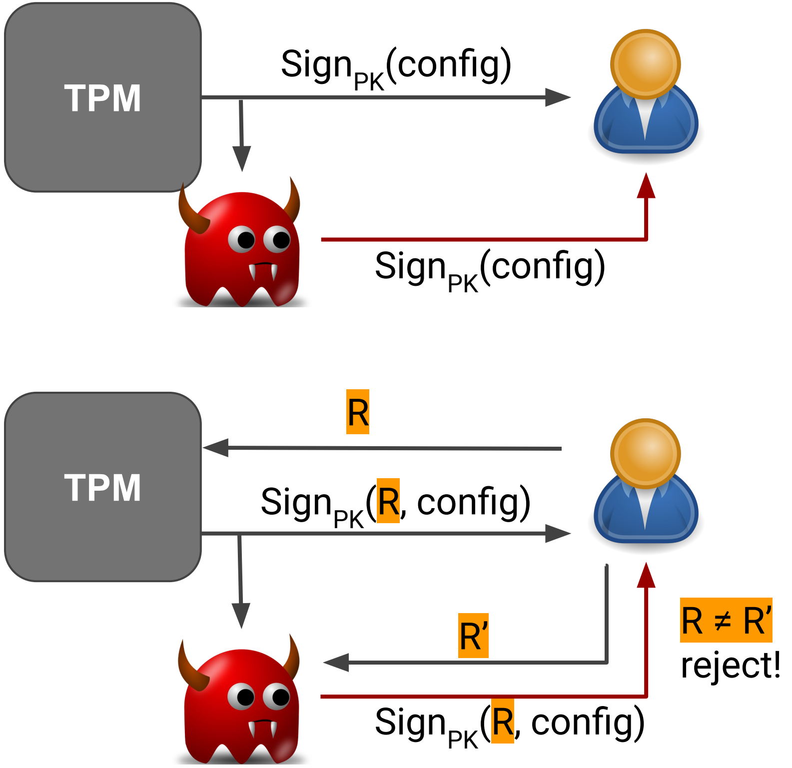
- TPM is **trustworthy** (no need of a further certification of the TPM)
- Only the TPM possesses this particular **private key**
- TPM's **public key** can be used to verify the signature
- **Hierarchical trust**: TPM certifies hardware/OS, OS can certify applications, etc.

Preventing replay attacks

An attacker might

1. **intercept** TPM certification
2. **compromise** the system
3. “**replay**” the certification when needed to prove trustworthiness of the attacked system

Solution: TPM includes a **random challenge** R from the requester in the signature to prevent “replays”



Encryption

Enables the **encryption of data** in such a way that the data can be decrypted only **by a certain machine**, and only if that machine is in a **certain (trusted) configuration**

Idea: one **master secret key** used to derive **many encryption keys**, one for each trusted configuration

⇒ decryption is possible only in the **same configuration**

Hierarchical trust: provide an encryption key to a (certified) application so that the application can encrypt data

Decryption can only be done by the **desired version** of the desired application running on the desired version of the desired OS

Even **remote**, if TPMs share master keys

Example: protected storage

File **encrypted** and saved in a local storage

The encryption key is **encrypted by the TPM** using the master key and stored together with the file

The encrypted key is associated to the specification of hardware / software configuration that is **authorized to access the key**

Application requests to decrypt the encrypted key:

1. TPM verifies that hardware / software **configuration** matches the required one
2. TPM **decrypts the key** and passes it to the application
3. Application decrypts the file and is **trusted to discard the key**