Principles of Computer Security

INF-744: Security and Privacy for IoT

Diego F. Aranha

Institute of Computing – University of Campinas

Introduction

Software is everywhere and performs critical functions:

- Energy and transport infrastructure;
- Economic activity;
- Government systems;
- Medicine;
- Electronic voting.

Different fields study the problem under different points of view, with some overlap:

- Software engineering;
- Dependability;
- Fault-tolerance;
- Formal methods;
- Information/Computer security.

Introduction

Heck, even different subfields of Information Security (InfoSec) study the problem under different points of view, with some overlap:

- Software Security
- Network Security (NETSEC)
- Operational Security (OPSEC)
- Cryptography (CRYPTO)
- Access Control
- Incident response
- Usability and human factors
- Privacy research
- Forensics and investigation
- Governance and legal issues

Important: Systems security = Software + Network security, Cybersecurity = mostly all of the above.

Introduction

Problem

Determining if a system is **secure**, that is, if it performs its functions correctly even under attack, is an **undecidable** problem.

Solution: Heuristic approach, based on accumulated experience, hoping to detect design and implementation errors.

Even with heuristic approach, still a challenging problem:

- Higher complexity, connectivity, extensibility;
- Tighter deadlines;
- Deep knowledge of the platform, programming language, support technologies, attack surface;
- Counter-intuitive nature of code;
- **Adversarial** reasoning (security mindset);
- Fundamental change of paradigm (culture).

Discussion: Do designer and attacker have asymmetric powers?

Scope

Definition

Security design is the art/science/engineering of designing, implementing and testing systems from the point of view of security, given a certain *threat model*.

Objectives of this course:

- Study best practices for secure design and implementation;
- Study classes of vulnerabilities and how attackers use them;
- Study protection mechanisms.

Important: Assume that attacker has access to details of the system design and implementation. If this is not the case, we are not studying security, but only **obfuscation**.

Preliminaries

System vulnerabilities can be split in two classes:

1. *Implementation errors*: low level, possible to detect automatically. Ex: memory corruption, race conditions.
2. *Design flaws*: high level, require manual analysis and treatment. Ex: key distribution, access control.

Design flaws and implementation errors create **risks** that the system will not fulfill its purpose. Risk is estimated as the product between the **probability** of exploitation and the corresponding **impact**.

Problem: human brain appears to be **bad** at risk assessment.

According to McGraw (2006), security design depends mainly on three factors:

1. Risk analysis and management;
2. Development process;
3. Knowledge of involved factors.

Risk analysis and management

Consists in the continuous process of **identifying**, **estimating**, **prioritizing** and **monitoring** risks generated by vulnerabilities during the system design and operation life cycle.

Composed by five fundamental activities:

1. Understanding context (purpose and functionality of the system);
2. Identifying risks and impacts;
   Examples of risks: system failure, unavailability, unauthorized access to data.
   Types of impact: financial, reputation, legislation, operating costs or development.
3. Combining and prioritizing risks by severity;
4. Defining mitigation strategies and operational procedures;
5. Applying and validating fixes.

Software development

Security should not be treated as an additional iteration at the end of the development process, but as a fundamental part of the design and implementation phases.

Example: After detecting plaintext messages are transmitted, implement SSL/TLS. But what about the keys, algorithms and certificates?

Steps of a secure development process:

- Training developers, architects and testers;
- Threat modeling;
- Defining security requirements;
- **Code review and architectural analysis**;
- Internal and **external** auditing of software components;
- Automated testing;
- Penetration testing;
- Generation of knowledge for the future.

Documentation

Threat model

Describes the **adversarial capabilities** (computing power, communication bandwidth) and **intervention points** available (attack surface, physical or remote access, intrusion depth).

Punchline: Security against who?

Documentation

Threat model

Describes the **adversarial capabilities** (computing power, communication bandwidth) and **intervention points** available (attack surface, physical or remote access, intrusion depth).

Punchline: Security against who?

Security requirements

Describes the **security properties** that must be provided by the software, given a certain adversarial model.

Example: Communication should be encrypted in transit.

Important: Documenting the process contributes to understanding clearly the adversary capabilities and security properties.

Analysis

Code review

Manual or automated detection of implementation errors. Beware of false positives! In many cases, helps to improve security, but seldom enough.

Important: Requires some adversarial reasoning.

Analysis

Code review

Manual or automated detection of implementation errors. Beware of false positives! In many cases, helps to improve security, but seldom enough.

Important: Requires some adversarial reasoning.

Architectural analysis

Manual detection of design flaws that prevent the system from fulfilling its security requirements. Emphasis should be put in the interfaces and data flows.

Important: Requires a lot of adversarial reasoning.

Important: Documenting fixes and solutions help validation.

Validation

Auditing

Code review and architectural analysis performed **internally** (by other members) or **independently** (by external members).

Important: Should never be performed by the original developers!

Validation

Auditing

Code review and architectural analysis performed **internally** (by other members) or **independently** (by external members).

Important: Should never be performed by the original developers!

Security testing

Following the test-driven development methodology, consists in verifying software behavior in exceptional cases, not only the expected functionality.

Important: Perfect to practice adversarial reasoning.

Validation

Auditing

Code review and architectural analysis performed **internally** (by other members) or **independently** (by external members).

Important: Should never be performed by the original developers!

Security testing

Following the test-driven development methodology, consists in verifying software behavior in exceptional cases, not only the expected functionality.

Important: Perfect to practice adversarial reasoning.

Penetration tests

Security testing performed by external members for exercising security mechanisms and unexpected scenarios.

Important: Take care when selecting candidates!

Development process

Final observations:

- Cheaper to detect design flaws and implementation errors in the **first** stages of development;
- Implementation errors are **amplified** by design flaws;
- Penetration tests are a **validation** step of secure design, but not the **entire** process;
- Secure software development requires alternating between **constructive** and **destructive** phases;
- Analysis should include dedicated security mechanisms (cryptography, access control);
- Security is not the sole responsibility of operators, but **mainly** of designers.

Knowledge

Principles, **practice**, **rules**, **vulnerabilities**, **attacks** and **history**:

- *Principle:* general wisdom accumulated through experience;
- *Practice:* recommendation of what to do and what to avoid;
- *Rule:* static definition of a practice, useful for automatic detection;
- *Vulnerability:* design and implementation issues that compromise a security property;
- *Attack:* method to exploit a vulnerability;
- *History:* security experience.

Security properties

Systems can have several different security properties:

1. **Confidentiality** or secrecy
2. **Integrity**
3. **Authenticity** or origin confirmation
4. **Non-repudiation**
5. **Availability**
6. **Reliability** or dependability
7. **Accountability**
8. **Anonymity**

In turn, these can be provided by different technologies:

1. Cryptography
2. Access control (identification, authentication, authorization)
3. Operational procedures

Principles

Least Privilege

Every program and user of a system should operate using the minimum set of privileges requires to fulfill the task, limiting the damage caused by abuse or accident.

Example: An e-mail server should only have privilege over its own files and should drop privileges as soon as additional tasks requiring them are finished.

Principles

Least Privilege

Every program and user of a system should operate using the minimum set of privileges requires to fulfill the task, limiting the damage caused by abuse or accident.

Example: An e-mail server should only have privilege over its own files and should drop privileges as soon as additional tasks requiring them are finished.

Defense in depth

For each security property, more than one security layer should be implemented. More importantly, vulnerabilities in one security layer should not amplify vulnerabilities in other security layers.

Principles

Least Privilege

Every program and user of a system should operate using the minimum set of privileges requires to fulfill the task, limiting the damage caused by abuse or accident.

Example: An e-mail server should only have privilege over its own files and should drop privileges as soon as additional tasks requiring them are finished.

Defense in depth

For each security property, more than one security layer should be implemented. More importantly, vulnerabilities in one security layer should not amplify vulnerabilities in other security layers.

Auditing or reconstruction

It should be possible to reconstruct the sequence of events that allows a successful attack (forensics).

Principles

Isolation or encapsulation

Software component should be isolated so that successful attacks are contained in a component and are not trivially extended to other components.

Example: Internal state accessible only through controlled interfaces.

Principles

Isolation or encapsulation

Software component should be isolated so that successful attacks are contained in a component and are not trivially extended to other components.

Example: Internal state accessible only through controlled interfaces.

Simplicity

Security comes from simplicity. Do not implement security mechanisms for software components which do not belong to the attack surface, to prevent **false sense of security**.

Principles

Isolation or encapsulation

Software component should be isolated so that successful attacks are contained in a component and are not trivially extended to other components.

Example: Internal state accessible only through controlled interfaces.

Simplicity

Security comes from simplicity. Do not implement security mechanisms for software components which do not belong to the attack surface, to prevent **false sense of security**.

Weakest link

Resources (time, costs) should be directed to the detection of attacks and protection at the attack surface points that offer the higher risks and compensation to the adversary, and not polishing already adequate security mechanisms.

Principles

Economy

Minimize the quantity of resources used by software components, anticipating the possibility that these resources could be future used by the adversary.

Principles

Economy

Minimize the quantity of resources used by software components, anticipating the possibility that these resources could be future used by the adversary.

Reusability

Do not replace a security-proven software component with a custom version, specially the components already available for the community to investigate.

Corollary: Do not design/implement your own crypto (unless you know what you are doing)!

Principles

Economy

Minimize the quantity of resources used by software components, anticipating the possibility that these resources could be future used by the adversary.

Reusability

Do not replace a security-proven software component with a custom version, specially the components already available for the community to investigate.

Corollary: Do not design/implement your own crypto (unless you know what you are doing)!

Transparency

Do not rely on the secrecy of the design and implementation details as a security measure. Simultaneously, minimize the information available to an attacker.

Principles

Elegant failure

The failure of any software component should conserve the security state of a system.

Example: A system performing multiple sensitive interdependent tasks should revert all steps in case any task fails.

Principles

Elegant failure

The failure of any software component should conserve the security state of a system.

Example: A system performing multiple sensitive interdependent tasks should revert all steps in case any task fails.

Reasonable defaults

Choose reasonable default and require manual intervention to change the configuration towards the extremes.

Principles

Elegant failure

The failure of any software component should conserve the security state of a system.

Example: A system performing multiple sensitive interdependent tasks should revert all steps in case any task fails.

Reasonable defaults

Choose reasonable default and require manual intervention to change the configuration towards the extremes.

Chain of trust

Trusted programs should not invoke or use non-trusted resources.

Practice

Good development practices from the security point of view:

- **Update** your knowledge to match the news and literature;
- Explore and learn from FLOSS;
- Handle untrusted inputs (command line, web forms) with extreme care (format and length);
- Dedicate special attention to the **secure storage** of information;
- Use common programming **patterns**;
- Remove obsolete code and **test** continuously;
- Assume that your users are **malicious**;
- Never assume success and **handle errors** correctly;
- Produce **records** of events captured by the software.

Attacks

There are many threats faced by computer systems:

1. **Backdoors:** inserted by design, attack or accident.
2. **Phishing:** targeted spoofing.
3. **Social engineering:** attacking our humanity.
4. **Cryptanalysis:** attacks against cryptography.
5. **Eavesdropping:** unauthorized access to communication channel.
6. **Spoofing:** impersonation of legitimate entity.
7. **Tampering:** malicious modification.
8. **Clickjacking:** user interface attacks.
9. **Denial of Service:** resource exhaustion.

Important: What security properties are being attacked in each?