# 1 Secure Software Principles

**Three pillars**
- Risk Management
- Touchpoints
- Knowledge

## 1.1 80% / 20% Problem Reduction

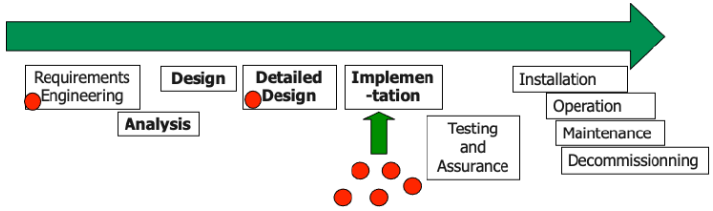
1. Secure the weakest link
2. Practice defence in depth
3. Fail securely
4. Follow the principle of least privilege
5. Compartmentalize
6. Keep it simple
7. Promote privacy
8. Remember that hiding secrets is hard
9. Be reluctant to trust (Off-the-shelf software)
10. Use your community resources (crypto algorithms)

# 2 Secure Software Lifecycle

**SW Engineering processes:** Waterfall, Spiral, Prototyping, XP, Adaptive Programming, ...
None of these really support security.

## 2.1 Software Engineering & Security

- Often ad hoc or not covered at all
- Issues:
  - Non systematic approach
  - Bad early decisions
  - Hard to manage and modify




## 2.2 Requirements / Analysis

- Identify security requirements
- Quantify the security risks
- Know the entire problem domain of the system

### 2.2.1 Identify Security Requirements

1. Identify stakeholders
2. Identify assets from different stakeholders (sensitive info / resources)
3. Identify requirements on these assets

**Result:** high-level security policy
**STRIDE:** Systematic approach for threat identification.
- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation disclosure
- **D**enial of service
- **E**levation of privilege

### 2.2.2 Quantify Security Risks

- Estimate and quantify the risk of the previously identified threats
  1. Importance or severity (critical vs. non-critical)
  2. Cost
  3. DREAD (Damage potential, Reproduceability, Exploitability, Affected users, Discoverability)
- Order the requirements and identify the relevant subset: Risk mitigation strategy

**Result:** high-level security policy revisited

### 2.2.3 Example

**General formulated:**
*The software must validate all user input to ensure it does not exceed the size specified fot that type of input.*
*The system must encrypt sensitive data transmitted over the Internet between the server and the browser.*
**Non-functional requirements:**

1. **Security Property Requirement**: specifies the characteristics that software must exhibit.
2. **Constraint or Negative Requirements** limit what software functionality can be allowed to behave.
3. **Security Assurance Requirements** are rules or best practices by which the software security functions will be built, deployed and operated.

## 2.3 Detailed Design

**Main goals:**

1. Identify security technologies that meet relevant security requirements
2. Bind these technologies to the application

### 2.3.1 Identify security technologies

- Select requirements that should be addressed at this level
- Identify security technologies that address requirements

**STRIDE Countermeasures: Tampering**
- Use data hashing and signing
- Digital signatures
- Strong authorization
- Tamper-resistant protocols across communication links
- Secure communication links with protocols that provide message integrity

**Information Disclosure**
- Strong authorization
- Strong encryption
- Secure communication links with protocols that provide message confidentiality
- Do not store secrets in plaintext

...

### 2.3.2 Bind Technologies to Application

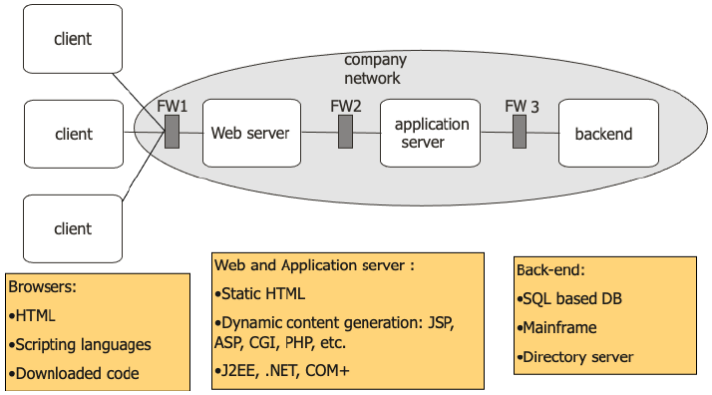
- For coarse-grained or simple requirements
  - Countermeasure can be realized in the operating system/middleware (SSL)
- For fine-grained and complex requirements
  - Implement as part of the application (hard to get right)

## 2.4 Other phases

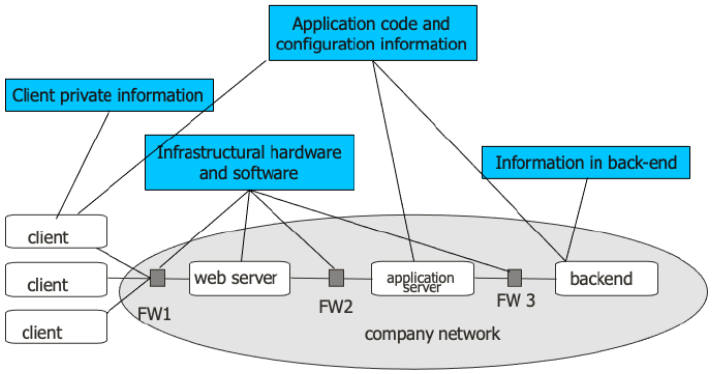
- Avoiding implementation vulnerabilities
- Security testing
- Automated patching

## 2.5 Case study: Web Applications

### 2.5.1 High level Architecture




### 2.5.2 Owners and Assets




### 2.5.3 Threat Agents and Threats

**Any hacker on the internet:**
- Spoof client or server
- Eavesdrop on connections / modify data in transit
- Bring down infrastructure components
- Gain unauthorized access to application or data

**Authorized user of the application: (additionally)**
- Repudiate transactions
- Elevate privilege

**Malicious server:**
- Steal private client information
- Spread spyware/viruses

### 2.5.4 Infrastructural countermeasures

**Authentication:**
- Network level: IPSEC
- Transport level: HTTP authentication mechanisms
- OS level: Windows authentication
- Single sing-on systems based on federation or windows active directory (Kerberos)
- Web authentication products: IAM products

**Data protection:**
- IPSEC (Network)
- Kerberos (OS)
- TLS (Transport)

**Access control:**
- Firewall
- In the webserver (URL)
- RBAC in the web container or application server
- File system based access control
- Access control products

**Sandboxing:**
- At the OS level: low-privileged accounts
- Language level: Java security architectures

**Others:**
- Filtering
- Throttling
- Shielding
- ...

### 2.5.5 Typical vulnerabilities

**Bugs in functional parts**
- Input validation
- Race conditions
- Bad error handling

**Broken countermeasures**
- Access control
- Authentication
- Crypto

### 2.5.6 Available Countermeasures at coding level

- Security technologies
- Quality improvements
  - Choice of programming language
  - Coding guidelines
  - Source code scanners
  - Security testing and audit
  - Code review

## 2.6 Daily Sins

1. Buffer Overruns
2. Format String problems
3. Integer Overflows
4. SQL Injection
5. Command Injection
6. Failing to Handle Errors
7. XSS
8. Failing to protect Network traffic
9. Use of magic URLs and Hidden Forms
10. Improper use of SSL and TLS
11. Use of Weak Password-Based systems
12. Failing to store and protect data securely
13. Information leakage
14. Improper file access
15. Trusting Network Name Resolution
16. Race Conditions
17. Unauthenticated Key Exchange
18. Cryptographically strong random numbers
19. Poor usability