Secure Development

DEF (*Security Properties*) Key security properties that must be considered in secure programming include:

1. **Confidentiality**: Data is only available to the people intended to access it.
2. **Integrity**: Data and system resources are only changed in appropriate ways by appropriate people.
3. **Availability**: Systems are ready when needed and perform acceptably.
4. **Authentication**: The identity of users is established (or you're willing to accept anonymous users).
5. **Authorization**: Users are explicitly allowed or denied access to resources.
6. **Nonrepudiation**: Users can't perform an action and later deny performing it.

DEF (*STRIDE*) STRIDE is a mnemonic for categories of security threats:

- **Spoofing**: Attacker pretends to be someone else.
- **Tampering**: Attacker alters data or settings.
- **Repudiation**: User can deny making an attack.
- **Information Disclosure**: Loss of personal information.
- **Denial of Service**: Preventing proper site operation.
- **Elevation of Privilege**: User gains higher privileges.

DEF (*Saltzer's Classic Principles*) Saltzer and Schroeder's principles provide guidelines for secure design:

1. **Economy of Mechanism**: Keep the design simple.
2. **Fail-Safe Defaults**: Default to secure configurations; fail closed with no single point of failure.
3. **Complete Mediation**: Check permissions on every access; ensure that all access requests are verified.
4. **Open Design**: Assume that attackers have access to the source code and specifications; design should not rely on obscurity.
5. **Separation of Privilege**: Require multiple conditions to grant access to sensitive operations; don't permit an operation based on a single condition.
6. **Least Privilege**: Grant only the minimum privileges necessary for users or processes; no more privileges than what is needed.
7. **Least Common Mechanism**: Minimize shared resources; be cautious of mechanisms that are shared among users.
8. **Psychological Acceptability**: Ensure that security measures are user-friendly and do not hinder usability; security should be intuitive for users.

DEF (*McGraw's Three Pillars*) McGraw's approach is built on three pillars:

1. **Applied Risk Management**: Identify, rank, and track risks using threat modeling to uncover security risks.
2. **Software Security Touchpoints**: Integrate security-related activities throughout the software development lifecycle.
3. **Knowledge**: Leverage existing knowledge, programming guidelines, and known exploits to enhance security practices.

DEF (*McGraw's Seven Touchpoints*) McGraw identified 7 touchpoints that could be integrated in the traditional software development lifecycle (SDLC)

1. **Abuse Cases**: Identify potential misuse scenarios.
   - Artifacts: Use case documents, threat models.
   - Problems: Failure to consider all misuse scenarios can lead to unaddressed vulnerabilities.
   - Bad Example: Ignoring input validation in a web application leading to SQL injection.
2. **Security Requirements**: Define security needs early in the process.
   - Artifacts: Security requirement specifications, risk assessment reports.
   - Problems: Vague or incomplete security requirements can result in inadequate protection.
   - Bad Example: A requirement stating "the application should be secure" without specifics on encryption or access controls.
3. **Architectural Risk Analysis**: Assess risks in design and architecture.
   - Artifacts: Architecture diagrams, risk analysis documents, design specifications.
   - Problems: Failing to identify security flaws in the architecture can lead to critical vulnerabilities.
   - Bad Example: Designing a system without considering the security of third-party components.
4. **Risk-Based Security Testing**: Focus on testing based on risk analysis.
   - Artifacts: Test plans, test cases, risk assessment matrices.
   - Problems: Inadequate testing of high-risk areas may leave critical vulnerabilities untested.
   - Bad Example: Conducting extensive tests on low-risk features while neglecting authentication mechanisms.
5. **Code Review**: Inspect code for security vulnerabilities.
   - Artifacts: Source code, code review checklists, static analysis reports.
   - Problems: Insufficient or poorly structured code reviews can miss significant vulnerabilities.
   - Bad Example: Relying solely on automated tools without manual review, leading to overlooked security issues.
6. **Penetration Testing**: Test the system for vulnerabilities in a real-world context.
   - Artifacts: Penetration testing reports, vulnerability assessment tools, exploitation scripts.
   - Problems: Conducting penetration tests without understanding the system can result in incomplete assessments.
   - Bad Example: External testers not familiar with the application's architecture, leading to ineffective testing.
7. **Security Operations**: Manage security during the deployment phase.
   - Artifacts: Security operation procedures, incident response plans, monitoring logs.
   - Problems: Lack of continuous monitoring can lead to undetected security incidents.
   - Bad Example: Deploying an application without proper logging and monitoring, making incident response difficult.

THM (*Dijkstra's Observation on Testing*) Dijkstra's famous remark states:

> "Testing shows the presence, not the absence of bugs."

This highlights the importance of comprehensive testing strategies and the understanding that passing tests do not guarantee that no bugs exist in the software.

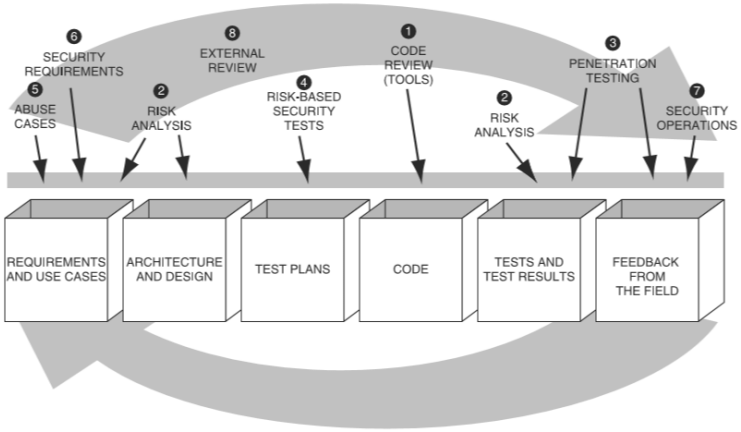

Figure 1: McGraw's Secure Software Development Lifecycle (SSDLC) emphasizes integrating security at various stages of software development.