**Chapter: Introductory class**

1. CIA Triad

**Confidentiality**

**Definition**: Ensuring that sensitive data is only accessible to those authorized to see it.

* ✅ **Goal**: Prevent unauthorized access.
* 🔒 **Methods**: Encryption, access controls, authentication, and classification of data.

**🧠 Example**:  
A company's HR database contains employee salaries. Access is restricted so only HR personnel can view them. If someone outside HR tries to access it and fails because of access controls, confidentiality is preserved.

**Integrity**

**Definition**: Ensuring that data is accurate, consistent, and hasn’t been tampered with.

* ✅ **Goal**: Maintain trustworthy data.
* 🔒 **Methods**: Hashing, digital signatures, checksums, and version control.

**🧠 Example**:  
An online banking transaction must not be altered during transfer. If Alice sends $100 to Bob, integrity ensures the amount isn’t changed to $1000 by a malicious actor. A cryptographic hash validates that the message hasn’t been changed.

**Availability**

**Definition**: Ensuring that systems and data are available to authorized users when needed.

* ✅ **Goal**: Minimize downtime.
* 🔒 **Methods**: Redundancy, backups, disaster recovery, and DDoS protection.

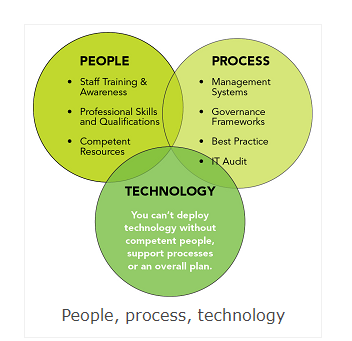
**🧠 Example**:  
An e-commerce site going down on Black Friday would cost millions. To ensure availability, the company uses load balancers, redundant servers, and cloud scaling to handle traffic and prevent outages.

**💡 Real-World Scenario: Online Banking System**

| **CIA Principle** | **Applied Example in Banking** |
| --- | --- |
| Confidentiality | Users must log in with 2FA to view account info. |
| Integrity | Transaction logs are hashed to detect tampering. |
| Availability | Redundant servers ensure 24/7 banking access. |

1. What is Cyber Security? Write about Three Pillars of Cyber Security.

Cyber security is the practice of defending computers, servers, mobile devices, electronic systems, networks, and data from malicious attacks.



1. What types of components are considering some (ICT Indices) organization to published the ranking?

|  |  |  |
| --- | --- | --- |
| Organization | Components | Ranking |
| Networked Readiness Index (NRI) | 1. Environment for ICT (market, political and regulatory, infrastructural environment), 2. The Stakeholders (individuals, businesses and government) to use ICT, and 3. The usage of ICT. | In 2024, Bangladesh is in 91th position. |
| ICT Development Index (IDI) | 1. ICT readiness (Infrastructure, access) 2. ICT Use (Intensity) 3. ICT capability (Skills) | In 2023, Bangladesh score is 61.1 |
| E-Government Development Index (EGDI) | 1. Online Service 2. Technological infrastructure 3. Human Capital | In 2024, Bangladesh is 111th position). |
| Global Cybersecurity Index (GCI) | 1. Legal measures 2. Technical measures 3. Organizational measures 4. Capacity development measures 5. Cooperation measures | In 2024, Bangladesh is 53th position). |
| National Cyber Security Index (NCSI) | 1. Legislation in force – Legal acts, regulations, orders, etc., 2. Established units – Existing organizations, departments, etc., 3. Cooperation formats – Committees, working groups, etc. and, 4. Outcomes – Policies, exercises, technologies, websites, programs, etc. | In 2024, Bangladesh s is 36th position. |

1. Dimensions of cybersecurity: Capacity maturity Model (CMM)
   1. Developing Cyber Security Policy & Strategy
   2. Encouraging responsible Cyber Security culture within society
   3. Building Cyber Security knowledge & capacity
   4. Creating effective legal & regulatory frameworks
   5. Controlling risks through standard & technologies

**Chapter: Introduction to Secure Software Development:**

1. Overview of secure software development principles

Secure software development involves embedding security throughout the Software Development Life Cycle (SDLC). This proactive approach ensures that security is not an afterthought but a core component of software design and implementation. Key principles include:

* **Security Requirement Engineering (SRE)**: Incorporating security requirements from the planning phase.
* **Secure Design Patterns**: Applying best practices like input validation, encryption, and proper authentication during the design phase.
* **Secure Coding Practices**: Reviewing code with a focus on security during development.
* **Dynamic Application Security Testing (DAST)**: Testing running applications to identify vulnerabilities.
* **Secure Deployment**: Ensuring that configurations are safe and secure at the deployment stage.
* **Continuous Monitoring and Maintenance**: Regular patching, auditing, and incident response planning post-deployment.

“Integrating security into the SDLC is not optional—it is a necessity in modern software development.”

1. Importance of integrating security into the software development life cycle (SDLC).

The slides highlight several reasons why security should be integrated into every phase of the SDLC:

1. **Early Detection and Mitigation of Vulnerabilities**
   * Fixing security issues early is far cheaper than post-deployment fixes.
   * Helps prevent critical flaws from reaching production.
2. **Reduced Cost of Fixes**
   * Addressing issues early avoids costly remediation, lawsuits, or data breach penalties.
   * Supported by industry reports (e.g., IBM’s Cost of a Data Breach).
3. **Enhanced Regulatory Compliance**
   * Helps meet strict industry regulations (e.g., GDPR, HIPAA, PCI-DSS).
   * Reduces the risk of non-compliance penalties.
4. **Protection Against Emerging Threats**
   * Involves proactive threat modeling and risk assessment.
   * Enables anticipation and mitigation of attack vectors.
5. **Improved Software Quality and Reliability**
   * Secure coding reduces bugs and performance issues.
   * Enhances system stability and resilience.
6. **Increased Customer Trust and Business Reputation**
   * Demonstrates commitment to user data protection.
   * Builds credibility and attracts more users.
7. How to integrate the security into the SDLCs?

­­­ To effectively integrate security into each phase of the SDLC, consider the following key practices:

1. Requirement Analysis (Planning Phase)

 Use Security Requirement Engineering (SRE) frameworks.

2. Design Phase

 Implement secure design patterns (e.g., input validation, encryption, authentication).

3. Development Phase

 Implement code reviews with security focus.

4. Testing Phase

 Perform Dynamic Application Security Testing (DAST) to detect vulnerabilities in running applications.

5. Deployment Phase

 Ensure secure configurations before deploying applications.

6. Maintenance and Monitoring Phase.

 Implement patch management and incident response plans.

 Conduct periodic security audits and compliance checks.

1. Describe Waterfall, Agile, DevOps model.

**Waterfall**

A linear and sequential model where each phase must be completed before moving to the next.

Phases:

1. Requirements Gathering

2. System Design

3. Implementation (Coding)

4. Testing

5. Deployment

6. Maintenance

Pros:

✔️ Simple and easy to understand

✔️ Well-structured with clear milestones

Cons:

❌ No flexibility for changes after the initial phase

❌ Late testing phase increases risk of defects

Best Used for:

 Small, well-defined projects Regulated industries (e.g., healthcare, banking)

**Agile**

A flexible and adaptive model where development is incremental and iterative, emphasizing customer collaboration.

Key Agile Frameworks:

 Scrum – Uses sprints (2-4-week development cycles)

 Kanban – Focuses on continuous workflow without predefined iterations

 Extreme Programming (XP) – Focuses on customer satisfaction, frequent releases, and

coding best practices

Pros:

✔️ Highly flexible to changes

✔️ Faster delivery with continuous feedback

Cons:

❌ Requires experienced developers

❌ Difficult to estimate cost and time

Best Used for:

 Start-ups, SaaS, mobile applications

 Projects with frequently changing requirements

**DevOps**

A modern SDLC approach that integrates development (Dev) and operations (Ops) for continuous integration and deployment (CI/CD).

Key Components:

 Continuous Integration (CI) – Automated code integration

 Continuous Deployment (CD) – Frequent and automated releases

 Infrastructure as Code (IaC) – Automated infrastructure management

Pros:

✔️ Faster software releases

✔️ Improved collaboration between teams

Cons:

❌ Requires strong automation and DevOps expertise

❌ Complex implementation in traditional organizations

Best Used for:

 Cloud applications, micro services, enterprise

**Chapter: Security Requirements Engineering:**

* 1. Identifying and defining security requirements.

Security Requirements Engineering (SRE) is the systematic process of identifying, documenting, and managing the security needs of a system. It aims to ensure protection against malicious attacks, misuse, and other security risks by addressing confidentiality, integrity, availability, and accountability.

**Key Activities:**

* + - **Stakeholder Analysis:** Identify key stakeholders (e.g., users, developers, regulators) and their security concerns.
    - **Threat Modelling:**
      * Use models like STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to detect potential threats.
    - **Risk Assessment:**
      * Analyse the impact and likelihood of identified threats.
    - **Specification:**
      * Document requirements using standard formats.
      * Use traceability links between threats and requirements.
      * Follow recognized standards such as ISO 27001 or NIST SP 800-53.
    - **Validation and Verification:**
      * Ensure alignment with stakeholder needs.
      * Confirm system implementation satisfies defined security needs through testing and audits.
    - **Example Security Requirements:**
      * Multi-factor authentication for admins.
      * AES-256 encryption for data at rest and in transit.
      * Role-based access control.
      * Logging and alerting for suspicious activities.
  1. Integrating security requirements with functional requirements.

Functional requirements define the specific behaviour or functions of a system. In contrast, (main features of the system)

Non-functional requirements specify how the system performs its tasks, focusing on attributes like performance, security, scalability, and usability.

Integration ensures that security is embedded into the system from the beginning and not treated as an afterthought.

**Key Practices:**

* + - Early Integration in SDLC:
      * Include security during the requirements-gathering phase.
      * Apply methodologies like Microsoft SDL (Security Development Lifecycle).
    - Mapping Security to Functional Needs:
      * Connect security controls to functionalities (e.g., login → secure authentication).
      * Define relevant non-functional requirements like encryption speed.
    - Security-by-Design:
      * Follow principles like least privilege and defence in depth.
      * Use patterns such as input validation or secure API design.
    - Collaboration Between Development & Security Teams:
      * Adopt DevSecOps to merge security into development pipelines.
      * Regularly perform static code analysis and penetration testing.
    - Continuous Monitoring and Improvement:
      * Implement security logging and monitoring.
      * Keep security measures updated against emerging threats.
  1. What is SRE? Write the importance, Example and Standard of SRE.

Security Requirements Engineering is the process of systematically identifying, documenting, and managing security-related requirements for a system.

**Importance:**

* Protects sensitive data and assets.
* Ensures compliance with legal, regulatory, and industry standards.
* Reduces the risk of security breaches and associated costs.
* Builds trust with stakeholders and users.

**Example:**

* Authentication: "The system shall require multi-factor authentication for all administrative users."
* Data Encryption: "All sensitive data shall be encrypted both at rest and in transit using AES-256 encryption."
* Access Control: "The system shall enforce role-based access control to restrict access to sensitive functions."
* Audit Logging: "The system shall log all access attempts and generate alerts for suspicious activity."

**Standards:**

Several standards and frameworks provide guidance on security requirements engineering:

* ISO/IEC 27001: Information security management.
* NIST SP 800-53: Security and privacy controls for federal information systems.
* OWASP ASVS (Application Security Verification Standard): Security requirements for web applications.
* Common Criteria (ISO/IEC 15408): Framework for specifying and evaluating security requirements.
  1. Write the Objectives and Steps of SRE.

**Objectives:**

* **Identify Security Risks:** Understand potential threats and vulnerabilities that could impact the system.
* **Define Security Goals:** Establish high-level security objectives, such as preventing unauthorized access or ensuring data integrity.
* **Specify Security Requirements:** Translate security goals into actionable and measurable requirements.
* **Integrate Security into the Development Lifecycle:** Ensure security is considered at every stage of the system development lifecycle (SDLC).

**Steps in SRE:**

The process of SRE typically involves the following steps:

* + - **Elicitation of Security Requirements**
      * Stakeholder Analysis: Identify stakeholders (e.g., users, developers, regulators) and their security concerns.
      * Threat Modelling: Use techniques like STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to identify potential threats.
      * Risk Assessment: Evaluate the likelihood and impact of identified threats.
    - **Analysis of Security Requirements**
      * Prioritization: Rank security requirements based on their criticality and the risks they address.
      * Conflict Resolution: Resolve conflicts between security requirements and other functional or non-functional requirements.
      * Feasibility Analysis: Ensure that the requirements are technically and economically feasible.
    - **Specification of Security Requirements**
      * Documentation: Clearly document security requirements using standardized templates or notations.
      * Traceability: Establish traceability links between security requirements, threats, and system components.
      * Use of Standards: Leverage industry standards (e.g., ISO 27001, NIST SP 800-53) to define security requirements.
    - **Validation and Verification**
      * Validation: Ensure that the security requirements align with stakeholder needs and security goals.
      * Verification: Confirm that the implemented system meets the specified security requirements through testing, reviews, and audits.

**Chapter: Threat Modelling:**

1. Techniques for identifying and assessing potential security threats.

The slides outline a **structured process** for identifying and assessing threats as part of **threat modelling**. The main techniques include:

#### 🔎 **1. Threat Identification Techniques**

Used to systematically discover possible threats that could impact the system:

* **STRIDE** (Developed by Microsoft):  
  Categorizes threats into six common types:
  + **S**poofing – Impersonation of a user/system
  + **T**ampering – Unauthorized data modification
  + **R**epudiation – Denial of performing an action
  + **I**nformation Disclosure – Unauthorized data access
  + **D**enial of Service – Disrupting system availability
  + **E**levation of Privilege – Gaining unauthorized access
* **Attack Trees**:  
  A visual technique where threats are structured like a tree, with the goal as the root and different ways to achieve that goal as branches. Helps analyze how attacks might be carried out.

#### 🧩 **2. Risk Assessment Techniques**

After identifying threats, assess them based on:

* **Likelihood** – How probable is the threat?
* **Impact** – What damage would it cause if it occurred?
* **Prioritization** – Rank threats based on risk to focus on high-impact/high-probability threats first.

#### 🧠 **3. System Modelling**

Used to visualize the system and its attack surface:

* **Data Flow Diagrams (DFDs)** – Represent how data flows through a system, showing inputs/outputs, processes, storage, and trust boundaries.
* **Architecture Diagrams** – Depict the structure of the system and components, which helps identify where security controls are required.

#### 🧰 **4. Threat Modelling Methodologies**

Different methodologies provide frameworks for assessing threats:

* **PASTA (Process for Attack Simulation and Threat Analysis)** – Focuses on aligning technical threats with business impact.
* **Trike** – Risk-based approach emphasizing attacker perspective.
* **VAST (Visual, Agile, Simple Threat Modelling)** – Designed for large-scale, agile environments.

#### 🛡️ **5. Validation and Review**

As part of the process:

* Continuously **validate** the threat model against the actual system.
* **Review** periodically to adapt to system changes and new emerging threats.

1. Creating threat models to analyse and prioritize risks.

We have to consider the following steps to creating threat models to analyse and prioritize risks.

**Step 1: Define the Scope**

* + Identify the system or application to be modeled.
  + Define the boundaries of the analysis, including what is in scope and out of scope.

**Step 2: Identify Assets**

* + Determine the critical assets that need protection, such as sensitive data, user credentials, or system functionality.

**Step 3: Create a System Model**

* + Develop a visual representation of the system, including its components, data flows, and trust boundaries.

Common modeling techniques include data flow diagrams (DFDs) and architecture diagrams.

**Step 4: Identify Threats**

* + Use threat identification techniques (e.g., STRIDE, attack trees) to enumerate potential threats to the system.
  + Consider both external and internal threats.

**Step 5: Assess Risks**

* + Evaluate the likelihood and impact of each identified threat.
  + Prioritize threats based on their risk level.

**Step 6: Mitigate Threats**

* + Develop strategies to address the identified threats, such as implementing security controls, redesigning components, or adding monitoring.

**Step 7: Validate and Review**

* + Validate the threat model to ensure it accurately represents the system and its risks.
  + Regularly review and update the threat model as the system evolves

1. What is Threat & Threat Modelling? Describe the Key concepts of threat modelling.

**Threat:** In Cyber Security threat is malicious activity to gain un authorized access.

**Threat Modelling:** Threat modelling is the structured approach to identifying, prioritizing and mitigating the security threats to the systems, applications and networks.

**Key Concepts of Threat Modelling:**

* + - Assets
    - Threat
    - Vulnerabilities
    - Attack surface
    - Mitigating

**Secure Architecture Design:**

1. Principles of designing secure software architecture

The core principles that guide secure software architecture design are:

**Defence in Depth (Layered Security)**

Apply multiple layers of security so that if one layer fails, others still protect the system.

Example: Combine firewalls, IDS, encryption, and access control.

**Least Privilege**

Grant the minimal necessary access to users and processes.

Example: Limit a database user to only needed tables and operations.

**Separation of Duties**

Distribute responsibilities to reduce risk and prevent misuse.

Example: Different people manage development and production.

**Fail-Safe Defaults**

Default to denying access unless explicitly allowed.

Example: Firewall rules block everything unless permitted.

**Economy of Mechanism**

Use simple, understandable, and minimal design for better security.

**Open Design**

Security should not depend on the secrecy of design; it should be effective even if the design is public.

**Psychological Acceptability**

Ensure security controls are user-friendly and do not hinder usability.

**Secure by Design**

Incorporate security from the start, not as an add-on.

Example: Threat modelling in early phases.

**Zero Trust Architecture**

Trust no device or user by default—even within the network.

Example: Micro-segmentation and constant verification.

**Regular Updates and Patch Management**

Keep systems updated to fix known vulnerabilities.

**Auditability and Logging**

Track all access and actions for analysis and incident response.

2. Security patterns and best practices for system design

Best practices and security patterns help implement the above principles effectively:

**Adopt a Secure Development Lifecycle (SDL)**

Integrate security throughout SDLC—include code reviews, audits, and pen testing.

**Implement Defence in Depth**

Apply multiple, independent protection layers such as WAFs, IDS, and encryption.

**Follow Least Privilege**

Ensure users, services, and processes have only the access they truly need.

**Use Strong Authentication and Authorization**

Apply MFA and use role-based or attribute-based access control.

**Encrypt Sensitive Data**

Use AES-256 or TLS 1.3 for encrypting data in transit and at rest. Secure your encryption keys.

**Validate and Sanitize Input**

Protect against injection attacks like SQLi and XSS by whitelisting and encoding input/output.

**Secure APIs**

Protect APIs with OAuth, API keys, and rate limiting. Always validate input/output.

**Monitor and Log Security Events**

Use centralized logging and SIEM tools for real-time monitoring and analysis.

**Regularly Update and Patch Systems**

Automate patching and perform regular vulnerability scans.

3.What is Secure Architecture Design?

Secure software architecture design is the process of creating software that is secure from cyber threats. It involves selecting and composing components, and designing systems and policies that protect business assets.

**Benefits of secure software architecture**

i) improves software design,

ii) Eliminates unnecessary access to resources,

iii) Streamlines processes,

iv) Reduces the chances of performance-degrading malicious activities, and;

v) Contributes to the overall quality of the software.

**Secure Coding Practices:**

1. Writing secure code and common programming vulnerabilities.

**Secure Coding:**

Secure coding is a structured approach to reduces the risk of vulnerabilities and helps ensure software integrity, confidentiality, and availability.

**Key reason for Secure Coding:**

* + - 1. Preventing Cyber Threats

Vulnerability Mitigation, Cybersecurity Incident Prevention & data protection

* + - 1. Building Trust and Maintaining Reputation

Customer Confidence & Reputation Enhancement

* + - 1. Cost and Time Savings

Reduced Security Incident Costs, Faster Development Cycles & Long-Term Maintainability

**Area of Secure Coding:**

* 1. **Authentication and Authorization:** Securely verifying user identities and granting appropriate access levels is fundamental.
  2. **Error Handling and Logging**: Implementing robust error handling and logging practices helps identify and address issues before they become major security vulnerabilities.
  3. **Input Validation and Sanitization:** Validating user inputs and sanitizing data before processing prevents malicious inputs from compromising the application.
  4. **Data Encryption:** Protecting sensitive data at rest and in transit through encryption ensures that even if the data is compromised, it remains unreadable.
  5. **Password Management**: Implementing strong password policies and secure storage practices minimizes the risk of credential compromise.
  6. **Threat Modelling**: Identifying and assessing potential threats to the application and implementing appropriate security controls is crucial for proactively building a secure application
  7. **Communication Security:** Ensuring secure communication channels and data transmission protocols protect against eavesdropping and manipulation.
  8. **Security Testing**: Regularly testing the application for security vulnerabilities helps ensure that the application remains robust and secure over time.
  9. **Mitigating Code Vulnerabilities:** Proactively addressing potential vulnerabilities in the codebase minimizes the risk of exploitation.
  10. **Education and Awareness:** Developers need to stay informed about the latest security threats and best practices.

**Common Programming Vulnerabilities**

Common programming vulnerabilities

Despite best efforts, developers often introduce vulnerabilities due to oversight or lack of awareness. Below are some of the most common vulnerabilities:

**1.Injection Attacks:**

SQL Injection: Occurs when untrusted input is used in SQL queries without proper sanitization, allowing attackers to manipulate databases.

Mitigation: Use parameterized queries or prepared statements.

Cross-Site Scripting (XSS): Occurs when malicious scripts are injected into web pages viewed by other users.

Mitigation: Sanitize user input and use frameworks that automatically escape output.

**2. Broken Authentication:**

Weak password policies, session hijacking, or improper session management can lead to unauthorized access.

Mitigation: Use strong password policies, secure session management, and multi-factor authentication.

**3. Sensitive Data Exposure:**

Storing or transmitting sensitive data (e.g., passwords, credit card numbers) without encryption.

Mitigation: Encrypt sensitive data at rest and in transit.

**4. Security Misconfiguration:**

Default configurations, unnecessary services, or unpatched software can expose systems to attacks.

Mitigation: Regularly review and harden configurations.

**5. Cross-Site Request Forgery (CSRF):**

Attackers trick users into performing actions they did not intend, such as changing account settings.

Mitigation: Use anti-CSRF tokens and validate requests.

**6. Buffer Overflows:**

Occurs when a program writes more data to a buffer than it can hold, potentially allowing attackers to execute arbitrary code.

Mitigation: Use safer programming languages (e.g., Python, Java) or functions that perform bounds checking.

**7. Race Conditions:**

Exploiting the timing of operations to gain unauthorized access or cause unintended behaviour.

Mitigation: Use synchronization mechanisms and atomic operations.

**8. Insecure APIs:**

Poorly designed or unprotected APIs can expose sensitive data or functionality.

Mitigation: Implement proper authentication, rate limiting, and input validation for APIs.

**Best Practice of Secure Coding:**

**1. Follow Secure Coding Guidelines:**

. Adhere to standards like OWASP Secure Coding Practices, CERT Secure Coding Standards, or language-specific guidelines.

**2. Use Security Libraries and Frameworks:**

Leverage well-tested libraries and frameworks for security functions (e.g., authentication, encryption).

**3. Conduct Security Testing:**

Perform regular security testing, including static analysis, dynamic analysis, and penetration testing.

**4. Educate Developers:**

Provide training on secure coding practices and common vulnerabilities.

**5. Implement Secure Development Lifecycle (SDL):**

Integrate security into every phase of the software development lifecycle, from design to deployment.

**6. Monitor and Respond to Threats:**

Continuously monitor applications for suspicious activity and respond to incidents promptly.

1. Code reviews and static code analysis for security.

**Code Reviews:**

A code review is a process where developers examine each other's code changes to improve code quality, identify potential errors, and ensure adherence to coding standards, ultimately leading to more reliable and maintainable software.

Benefits of Code Reviews for Security

1. Early Detection of Vulnerabilities:

2. Knowledge Sharing:

3. Improved Code Quality:

4. Compliance with Standards:

**Types of Code Reviews:**

1. Formal Reviews:

Structured and scheduled reviews with a defined process and checklist.

2. Informal Reviews:

Ad-hoc reviews, such as pair programming or quick peer reviews.

3. Tool-Assisted Reviews:

Use tools to automate parts of the review process, such as identifying syntax errors or style violations.

**Static Code Analysis**

**Purpose of Static Code Analysis**

1. Detect Bugs and Vulnerabilities

2. Enforce Coding Standards

3. Improve Code Maintainability

4. Provide Early Feedback

**Types of Static Code Analysis**

1. Syntax Checking:

Ensures the code follows the language's syntax rules.

2. Style Checking:

Enforces coding style guidelines (e.g., indentation, naming conventions).

3. Bug Detection:

Identifies common programming errors (e.g., uninitialized variables, dead code).

4. Security Analysis:

Detects vulnerabilities such as injection flaws, insecure API usage, and hardcoded credentials.

5. Performance Analysis:

Highlights code that may cause performance issues (e.g., inefficient loops, memory leaks).

**Limitations of Static Code Analysis**

1. False Positives: Tools may flag issues that are not actual vulnerabilities, requiring manual verification.

2. False Negatives: Some vulnerabilities may not be detected, especially if they involve complex logic or runtime behaviour.

3. Limited Context: Static analysis cannot account for the dynamic behaviour of the application.

**Differences Between Code Reviews and Static Code Analysis**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Code Reviews** | **Static Code Analysis** |
| **Approach** | Manual, human-driven process | Automated, tool-driven process |
| **Scope** | Broader, including design and logic | Focused on code patterns and syntax |
| **Execution** | Requires human effort and collaboration | Runs automatically without human intervention |
| **Strengths** | Detects complex logic and design issues | Identifies syntax errors and common bugs |
| **Limitations** | Time-consuming and subjective | May produce false positives or negatives |
| **Tools** | Pull request tools (e.g., GitHub, GitLab) | Static analysis tools (e.g., SonarQube, Checkmarx) |