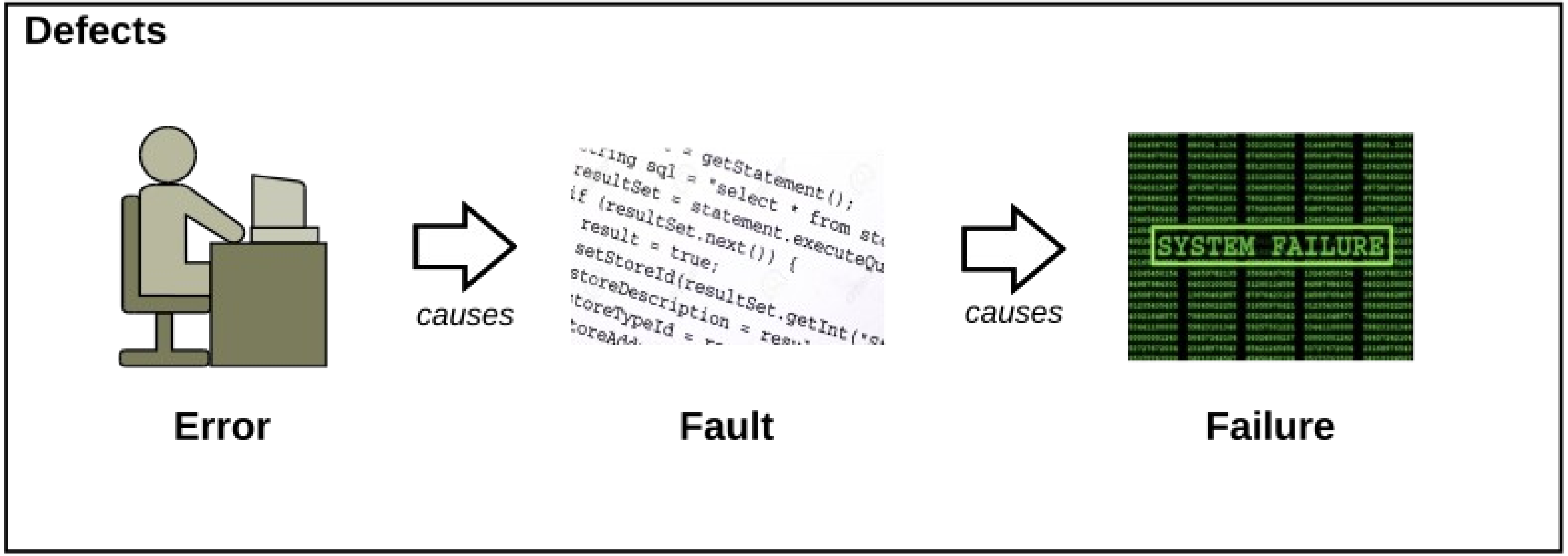
Secure Code

Review



Error: a human action that eventually leads to a fault

Fault: an incorrect step in building the system at any point that results in failure

Failure: any place the software does not perform as required

Defect: a generic term for any of the above

Robust Software

- Software is robust when it has
  - "the ability to cope with errors during execution and to handle erroneous input"
- Three types of robustness
  - Safe: when the system can detect, respond to or prevent accidental harm
  - Secure: when the system can detect, respond to or prevent intentional harm
  - Survivable: when the system is both safe and secure

Software Engineering

- Focuses on eliminating defects
  - To remove any faults that prevent the software from working as specified
  - To ensure the software handles the normal and reasonable situations and inputs correctly, including invalid inputs
- Does not focus on intentional attacks
  - Attacks usually involve attempting to put the system into an abnormal situation or unusual state
  - Attacks also usually involve bizarre, unreasonable and highly unusual inputs
  - Not the type of inputs that would the thought of when looking at normal operations
  - Also, the inputs my occur with a volume and velocity that would stress the system
  - The imposed stress would cause the system to go into an unstable state

Security Engineering

- A security flaw is
  - A defect in or a feature of the software that can be exploited by an attacker
  - A defect that is fixed for normal operations (i.e. safe) may still be a security flaw
  - Not all defects are security flaws
  - Only defects that can be exploited are security flaws
- A vulnerability is
  - A set of circumstances that allow an attacker to exploit a security flaw

Security Engineering

- A mitigation is the removal of a vulnerability either
  - By fixing the underlying security flaw; or
  - Developing a workaround that prevents attackers from accessing the security flaw
- Not all security flaws can be fixed
  - The cost of fixing the flaw may be prohibitive
  - The flaw may be complex or involve multiple components which means it may be a systemic problem not a defect

STRIDE Attack Definitions

- Microsoft’s model for identifying threats in software
- STRIDE is an acronym for categorizing attacks
  - Spoofing: Pretending to be something or someone else
  - Tampering: Unauthorized modification of anything in a system or application
  - Repudiation: Denying responsibility for something
  - Information Disclosure: Providing information to unauthorized parties
  - Denial of Service: Making system resources unavailable for use
  - Elevation of Privilege: Performing actions that are not authorized
- Helps identify potential threats early in the design and development process.

S - Spoofing

- Spoofing
  - Definition: Pretending to be someone/something else.
  - Impact: Unauthorized access to systems.
- Examples:
  - Java: An attacker forges a JWT token and bypasses Spring Security filters.
  - Python: Fake login cookies accepted by a Flask app.
- Mitigation:
  - Strong authentication (MFA, strong passwords).
  - Use signed tokens (JWT with proper secret/key).
  - Never trust client-supplied identity data.

T – Tampering with Data

- Tampering with Data
  - Definition: Unauthorized modification of data at rest or in transit.
  - Impact: Corrupted data, altered transactions.
- Examples
  - Java: Modifying serialized objects before deserialization.
  - Python: Man-in-the-middle alters API request data.
- Mitigation:
  - Digital signatures and checksums.
  - TLS (secure sockets) for secure transport.
  - Avoid unsafe deserialization like the Java default serialization/deserialization

R – Repudiation

- Repudiation
  - Definition: Ability of users to deny performing an action without detection.
  - Impact: Lack of accountability, difficulty in audits.
- Examples
  - Java web service without proper logging → attacker deletes records and denies it.
  - Python Flask app logs only user IDs but not timestamps.
- Mitigation
  - Implement tamper-proof logging (append-only, signed).
  - Correlate logs with unique request IDs.
  - Apply non-repudiation mechanisms (e.g., digital signatures).

I – Information Disclosure

- Information Disclosure
  - Definition: Exposure of information to unauthorized parties.
  - Impact: Loss of confidentiality, data leaks.
- Examples:
  - Java stack traces displayed in production, leaking DB schema.
  - Python app logs secrets (API keys) in error messages.
- Mitigation:
  - Suppress verbose error messages in production.
  - Sanitize logs (no passwords/tokens).
  - Encrypt sensitive data at rest and in transit.

D – Denial of Service (DoS)

- Denial of Service
  - Definition: Making a system unavailable to legitimate users.
  - Impact: Service disruption, downtime, financial loss.
- Examples:
  - Python: Expensive regex (re catastrophic backtracking).
  - Java: Uploading extremely large files to exhaust memory.
- Mitigation:
  - Input throttling and rate limiting.
  - Use timeouts and circuit breakers.
  - Monitor unusual spikes in requests.

E – Elevation of Privilege

- Elevation of Privilege
  - Definition: Gaining higher permissions than authorized.
  - Impact: Attackers gain admin/root access.
- Examples:
  - Java web app where normal users access /admin endpoints due to misconfigured access controls.
  - Python app using os.system("rm -rf " + user_input) allowing arbitrary command execution.
- Mitigation:
  - Enforce least privilege (users get only what they need).
  - Perform strict input validation before executing system commands.
  - Use role-based access control (RBAC).

Security: Preventive Planning

- Design with the objective that the API will eventually be accessible from the public internet
  - Even if there are no immediate plans to do so
- Use a common authentication and authorization pattern, preferably based on existing security components
  - Avoid creating a unique solution for each API
- Least Privilege
  - Access and authorization should be assigned to API consumers based on the minimal amount of access they need to carry out the functions required

Security: Preventive Planning

- Maximize entropy (randomness) of security credentials
  - Use API Keys rather than username and passwords for API
- Balance performance with security with reference to key lifetimes and encryption/decryption overheads
- Standard secure coding practices should be integrated
  - More on this later
- Security testing capability is incorporated into the development cycle
  - Continuous, repeatable and automated tests to find security vulnerabilities in APIs and web applications during development and testing

Security: Use CVE

- CVE = Common Vulnerabilities and Exposures.
  - An international, community-driven effort that identifies and catalogs publicly known cybersecurity vulnerabilities.
  - Each vulnerability is assigned a unique CVE ID (e.g., CVE-2024-12345).
  - Managed by the CVE Program,
    - *Overseen by MITRE Corporation*
    - *Sponsored by the U.S. Department of Homeland Security (DHS CISA).*
- Goals of CVE
  - Provide a single, standardized identifier for vulnerabilities.
  - Eliminate confusion caused by multiple vendors using different names for the same issue.
  - Enable security tools, databases, and services to reference vulnerabilities consistently.
  - Serve as the foundation for related resources like the NVD (National Vulnerability Database).

Security: Use CVE

- How CVE IDs Are Assigned
  - A researcher or vendor finds a vulnerability.
  - They request a CVE ID from a CVE Numbering Authority (CNA) (e.g., Microsoft, Red Hat, Apache, or MITRE).
  - Once confirmed, the vulnerability is published with its CVE ID.
- Example CVE Record
  - CVE-2023-4863
  - Description: A heap buffer overflow in the WebP image library (libwebp).
  - Impact: Remote code execution when processing malicious images.
  - References: Links to Google advisory and patches.
  - Status: Published.

Security: Use CVE

- How Developers & Engineers Should Use CVE
  - Monitor: Stay aware of new vulnerabilities in software you use.
  - Use CVE feeds or vendor advisories.
  - Assess Risk: Cross-check with NVD for CVSS severity ratings.
  - Patch: Apply vendor updates or mitigations as soon as possible.
  - Document: Track CVEs relevant to your systems for compliance reports.
  - Integrate: Use automated tools (e.g., pip-audit for Python, OWASP Dependency-Check for Java/Maven) that map library vulnerabilities to CVE IDs.

Common CVE Scanning Tools

- For Python
  - pip-audit (by PyPA)
    - *Scans Python environments and project dependencies.*
    - *Maps vulnerabilities to CVE IDs.*
    - *Example: pip-audit -r requirements.txt*
  - Safety (by PyUp)
    - *Checks Python packages for known vulnerabilities.*
    - *Database references CVEs and advisories.*
- For Java / JVM
  - OWASP Dependency-Check
    - *Supports Maven, Gradle, and other ecosystems.*
    - *Identifies dependencies with known CVEs.*
    - *Integrates with CI/CD pipelines.*
  - Snyk (supports Java, Python, Node, etc.)
    - *Cloud-based with free tier.*
    - *Provides CVE mapping, severity, and remediation advice.*

Common CVE Scanning Tools

- For Source Code Repos
  - GitHub Dependabot
    - *Automated dependency scanning in GitHub projects.*
    - *Creates PRs to fix vulnerabilities (mapped to CVEs).*
- GitLab Dependency Scanning
  - Similar integration for GitLab CI/CD pipelines.

Authentication and Authorization

- Authentication
  - Uses agent's information to identify them
  - Verifies the agent's credentials
  - Must occur before any authorization happens
  - Confirming the truth of some piece of data used by agent to identify themselves
- "How can you prove who you are?"

Authentication and Authorization

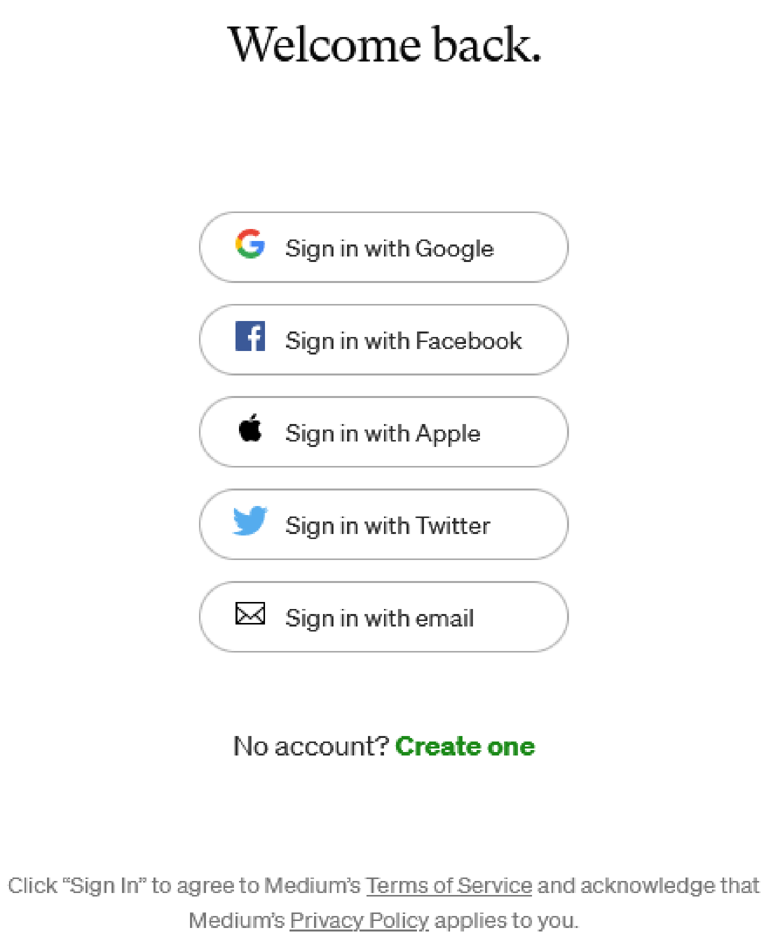
- Authorization
  - Checks an agent’s right to access a resource
  - Validates the agent’s permissions
  - Occurs after the identity of the agent is confirmed
  - Specifies the rights, permissions and privileges of an authenticated agent
- “How do we know what you are allowed to do?”

Password Fatigue

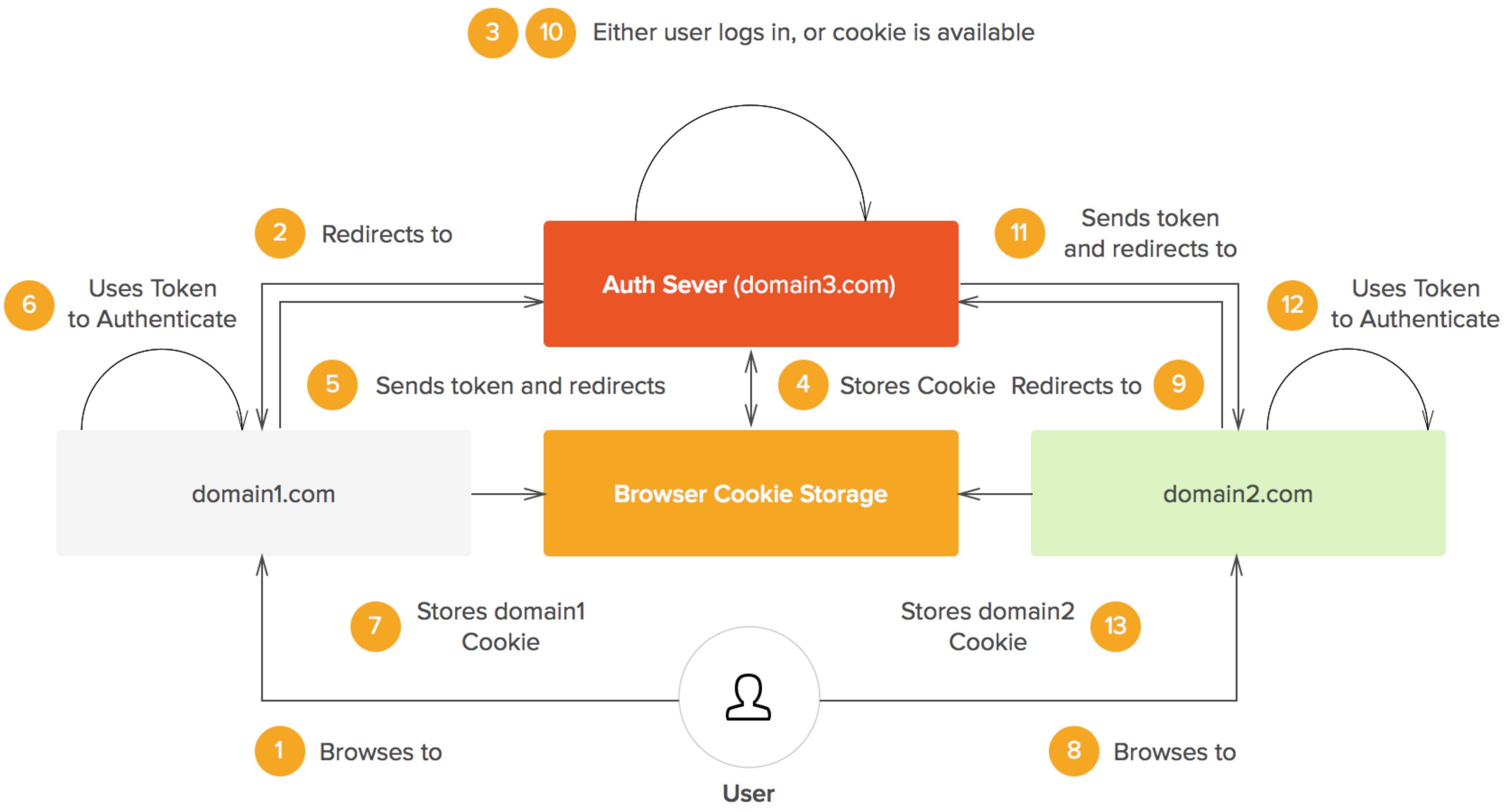
- Feeling experienced by managing too many user ids and passwords
- Creates a social engineering security risk
  - Users use the same password everywhere – a security vulnerability
  - Users do not change their passwords regularly
  - Users tend to use easily remembered (easily cracked) passwords
  - Users tend to record passwords and account information insecurely
- The various authentication credentials used are called “secrets”
  - A main security vulnerabilities is poor secrets management

Single Sign-On

- User can log in with a single ID and password to multiple systems
- Authentication is shared between the systems
- The systems are independent but are related in some way
- Also referred to as a federated login across networks



Identity Broker and SSO



Encryption

- Symmetric Encryption
  - Definition: Uses the same key for both encryption and decryption.
  - Strengths: Fast, efficient for large amounts of data.
  - Weaknesses:
    - *Key distribution is difficult*
    - *Both sender and receiver must share the same secret key securely.*
  - Algorithms: AES, DES, ChaCha20.

Encryption

- Asymmetric Encryption
  - Definition: Uses a key pair – a public key and a private key.
    - *Public key: shared openly, used to encrypt.*
    - *Private key: kept secret, used to decrypt.*
  - Strengths: Solves key distribution problem; supports digital signatures.
  - Weaknesses:
    - *Slower than symmetric encryption*
  - Usually combined with symmetric methods in practice (e.g., SSL/TLS).
    - *Message is encrypted with symmetric encryption*
    - *Key is encrypted using asymmetric encryption*
- Use Cases:
  - Secure key exchange (e.g., establishing an AES session key).
  - Digital signatures for authenticity and non-repudiation.

Encryption

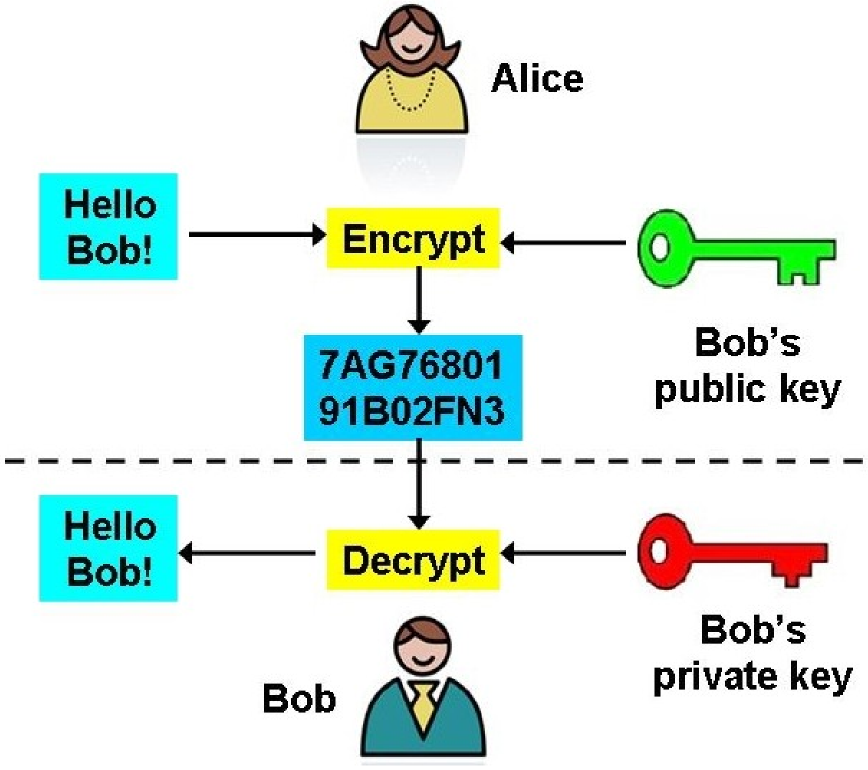
- Hashing (One-Way Functions)
  - Definition: Irreversible mathematical function → same input always gives same output.
- Properties:
  - Deterministic but one-way (cannot recover input).
  - Collision-resistant (hard to find two different inputs with same hash).
- Use Cases:
  - Password storage (with salt & stretching).
  - Integrity checks (file verification).
  - Algorithms: SHA-256, SHA-3, bcrypt, PBKDF2, Argon2.

Salting and Stretching

- Salting
  - A salt is a random string that gets added to a password before hashing.
  - Ensures that the same password does not result in the same hash.
  - Prevents the use of rainbow tables (precomputed hash lookups).
  - Makes each hash unique, even if users pick identical passwords.
- Stretching
  - Stretching means making the hashing process computationally expensive by repeating or slowing down the hash calculation.
- Purpose:
  - Slows down brute-force attacks (attackers must spend more CPU/GPU time per guess).
  - Even if an attacker gets the hashed database, cracking becomes impractical.
- Techniques:
  - PBKDF2 (Password-Based Key Derivation Function 2) – iterates hashing thousands of times.
  - bcrypt – automatically salts and repeats internally, adjustable cost factor.
  - Argon2 – modern, memory-hard algorithm designed to resist GPU/ASIC cracking.
- Example:
  - A single SHA256 hash takes microseconds → attacker can try billions of guesses per second.
  - A bcrypt hash with cost=12 might take 300ms → attacker slowed to a few guesses per second.

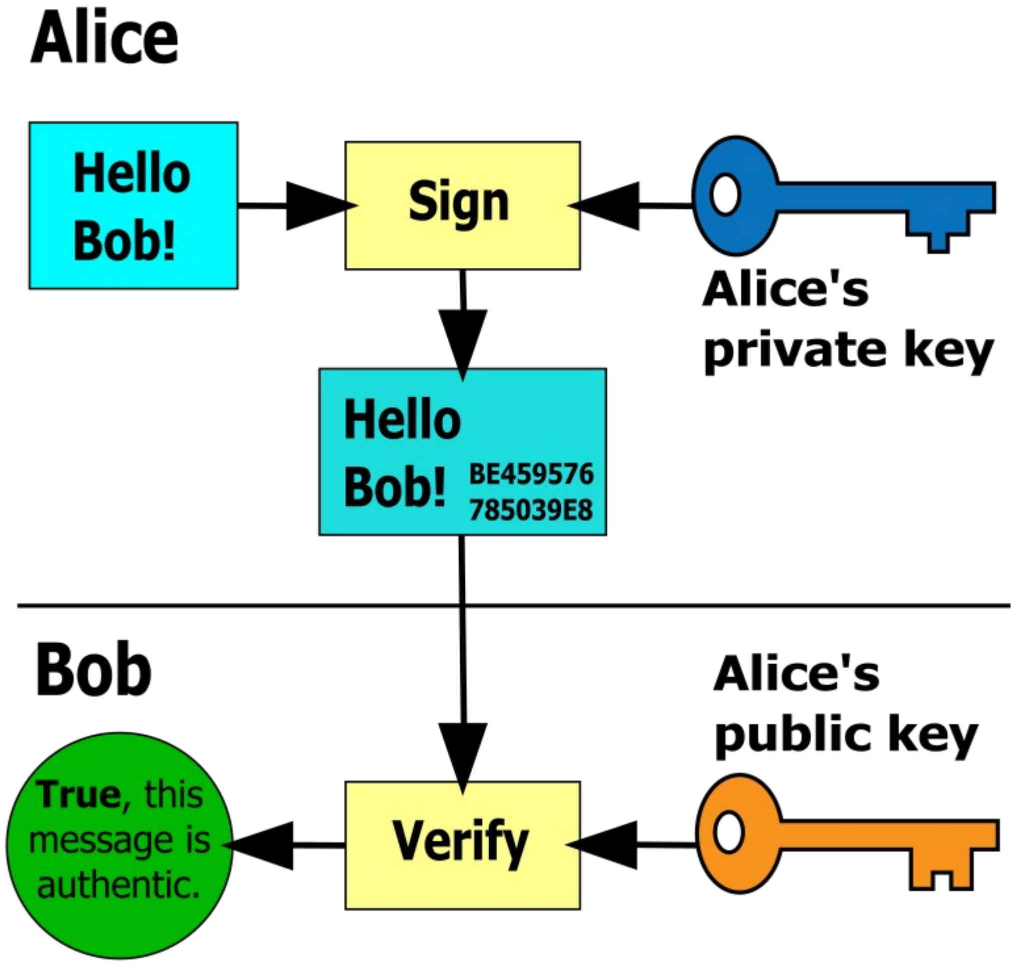
Encryption Uses

- Uses a public/private key pair
  - The public key can encrypt text sent to the key owner
  - Only the key owner's private key can decrypt the cipher text
  - The public key cannot decrypt



Encryption Uses

- Digital Signatures
- To sign a message
  - A hash the message is made
  - Then encrypted with a private key
  - This is the digital signature
  - Only the owner of the private key can create a signature
- Verification
  - The signature is decrypted with the sender's public key
  - The decrypted hash is compared to a new hash of the message
  - A match = verified authentic

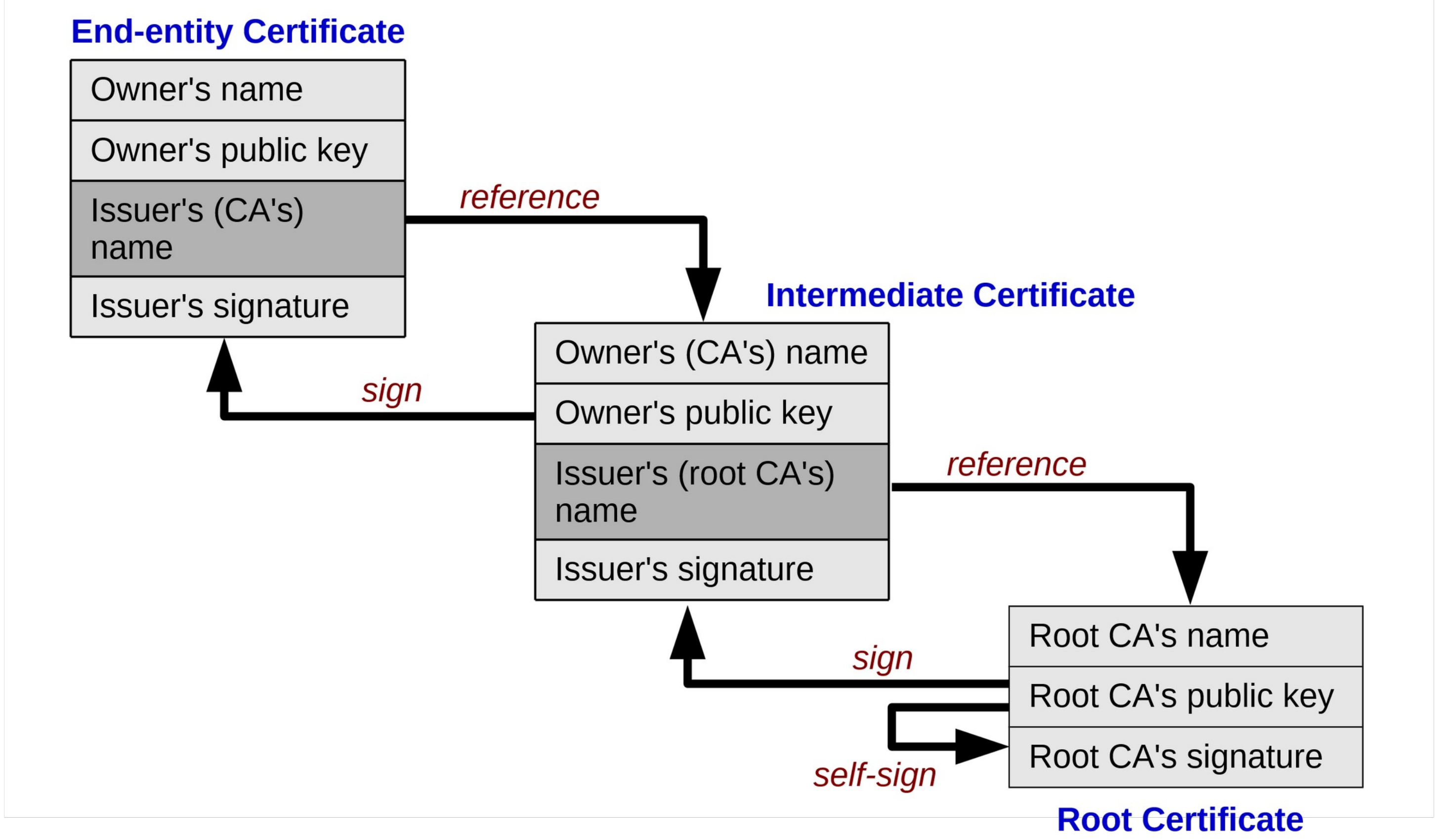


Certificates and Trust

- An X509 digital certificate is a cryptographic ID document
  - My certificate is used to verify my identity
  - Issued by a CA or certificate authority
  - The CA signs my certificate with their private key to verify it is really mine
  - The CA signed certificate acts a trusted third party that has vouched for me
- The CA’s certificate is signed by another CA
  - The chain of CA signatures starts with a root certificate or trust anchor
  - This establishes a “chain of trust” - signatures can be verified

Certificates and Trust

- Every CA must meet strict requirements and undergo a compliance audit
  - There are about 50 trusted root CAs

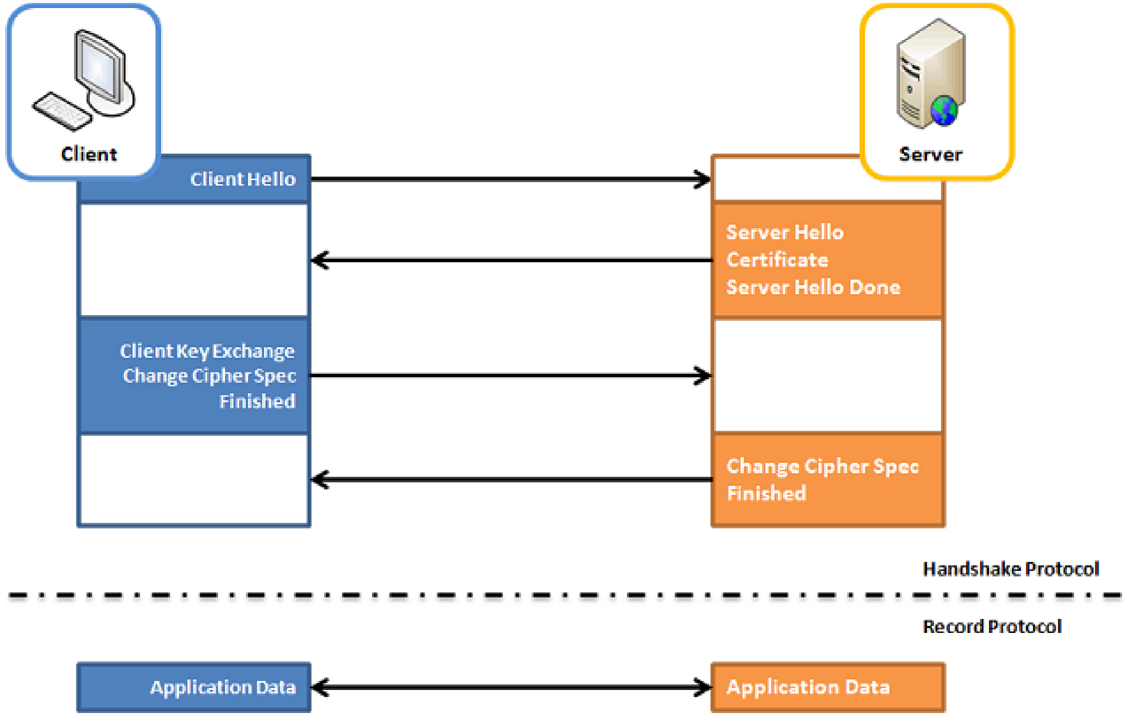


TSL – Transport Security Layer

- Cryptographic protocol
  - End-to-end security of data sent between applications over the Internet
  - Used to establish secure browser sessions with HTTPS
  - Also used for email, video/audio conferencing, IM, VOIP, and other services
- Implementation of security in transit imperative
  - Information in transit is secure from eavesdropping or tampering
  - Does not ensure security at rest
  - Information may be compromised at either before or after transmission
  - In cases where the identity of the server is not in question
  - Self signed certificates may be used (most browsers will warn about this)

TSL – Transport Security Layer

- Starts with a “handshake”
  - Certificate is given to the client to verify the server ID during the session
  - Asymmetric keys are created for the session
  - Session keys are used to encrypt the data in transit



SSL authentication handshake messages

NIST Secure Coding Standards

- NIST SP 800-218 (SSDF)
  - SSDF = Secure Software Development Framework
  - Published by NIST (National Institute of Standards and Technology)
  - A high-level framework for integrating security into the software development lifecycle (SDLC).
  - Built to be technology-agnostic, but maps to specific coding practices (e.g., Java, Python).

Key Principles (SSDF Practices)

- Define Security Requirements Early
  - Include security in functional and design requirements.
  - Example: “All passwords must be hashed using PBKDF2/bcrypt with salt.”
  - Avoid retrofitting security after code is written.
- Implement Secure Coding Guidelines
  - Follow published standards (e.g., CERT, OWASP).
  - Use secure defaults in frameworks.
  - Example: Disable weak cipher suites in Java SSL context.

Key Principles (SSDF Practices)

- Verify with Automated Tools and Peer Reviews
  - Static Analysis (SAST): e.g., SonarQube, Bandit (Python), SpotBugs (Java).
  - Dependency Scanners: pip-audit, OWASP Dependency-Check.
  - Peer Reviews: enforce security checklists during code reviews.
- Monitor & Respond Post-Deployment
  - Log security-relevant events (logins, privilege changes).
  - Monitor CVEs for dependencies.
  - Apply patches quickly.
  - Example: Using pip-audit to check for Python package CVEs weekly.

CERT Secure Coding Standards

- Developed by CERT/SEI (Carnegie Mellon University).
  - Provides language-specific secure coding rules for: Java, C / C++, Perl, Android, etc.
  - Rules are categorized as MUST, SHOULD, or CONSIDER.
  - Example
    - *Java: “Do not expose sensitive data in exceptions or logs.”*
  - The standard gives examples of insecure code and corrected secure code
  - Excellent reference for securing code

CERT Secure Coding Standards

- CERT uses a classification of rules and recommendations
  - To help organizations measure how thoroughly they are applying the standards
  - These categories often serve as a practical compliance ladder.
- Mandatory requirements.
  - Violations of rules are considered unacceptable because they can lead to exploitable vulnerabilities.
    - *Example (Java rule): EXP00-J – Do not expose sensitive data in exceptions.*
  - Compliance meaning: All rules must be followed for full compliance.

CERT Secure Coding Standards

- Recommendations
  - Guidance that should be followed whenever practical.
  - Violations don't always introduce immediate security risks but may reduce robustness or increase attack surface.
    - *Example (Java recommendation): NUM07-J – Use integer types with sufficient range to prevent overflow.*
  - Compliance meaning: A codebase that follows all rules and most recommendations is considered highly compliant.
- Considerations
  - Advice on good practices, coding style, or architectural preferences.
  - They are optional and provide additional guidance for developers aiming at the highest level of secure coding maturity.
    - *Example: Using immutable objects where possible in Java for thread safety.*
  - Compliance meaning: Following considerations is not required but demonstrates maturity beyond compliance.

CERT Secure Coding Standards

- Assessment
  - Potential risk of not meeting a rule or recommendation

Severity—How serious are the consequences of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	Low	Denial-of-service attack, abnormal termination
2	Medium	Data integrity violation, unintentional information disclosure
3	High	Run arbitrary code

Likelihood—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

Value	Meaning
1	Unlikely
2	Probable
3	Likely

Remediation Cost—How expensive is it to comply with the rule?

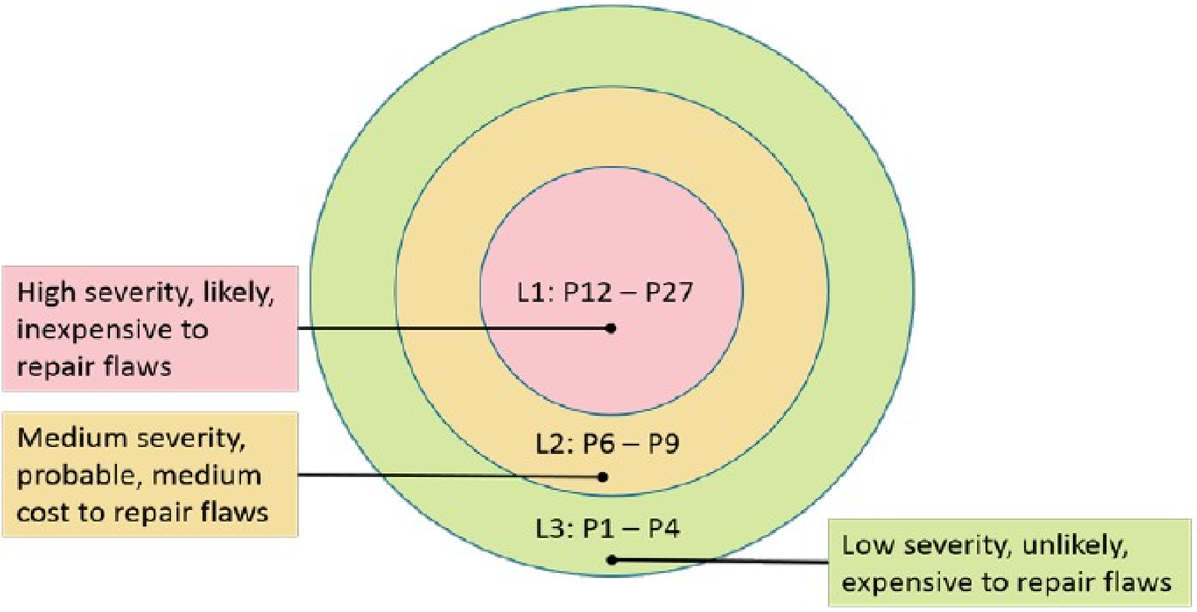
Value	Meaning	Detection	Correction
1	High	Manual	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

CERT Secure Coding Standards

- Rating
  - Combined risk analysis from the previous slide

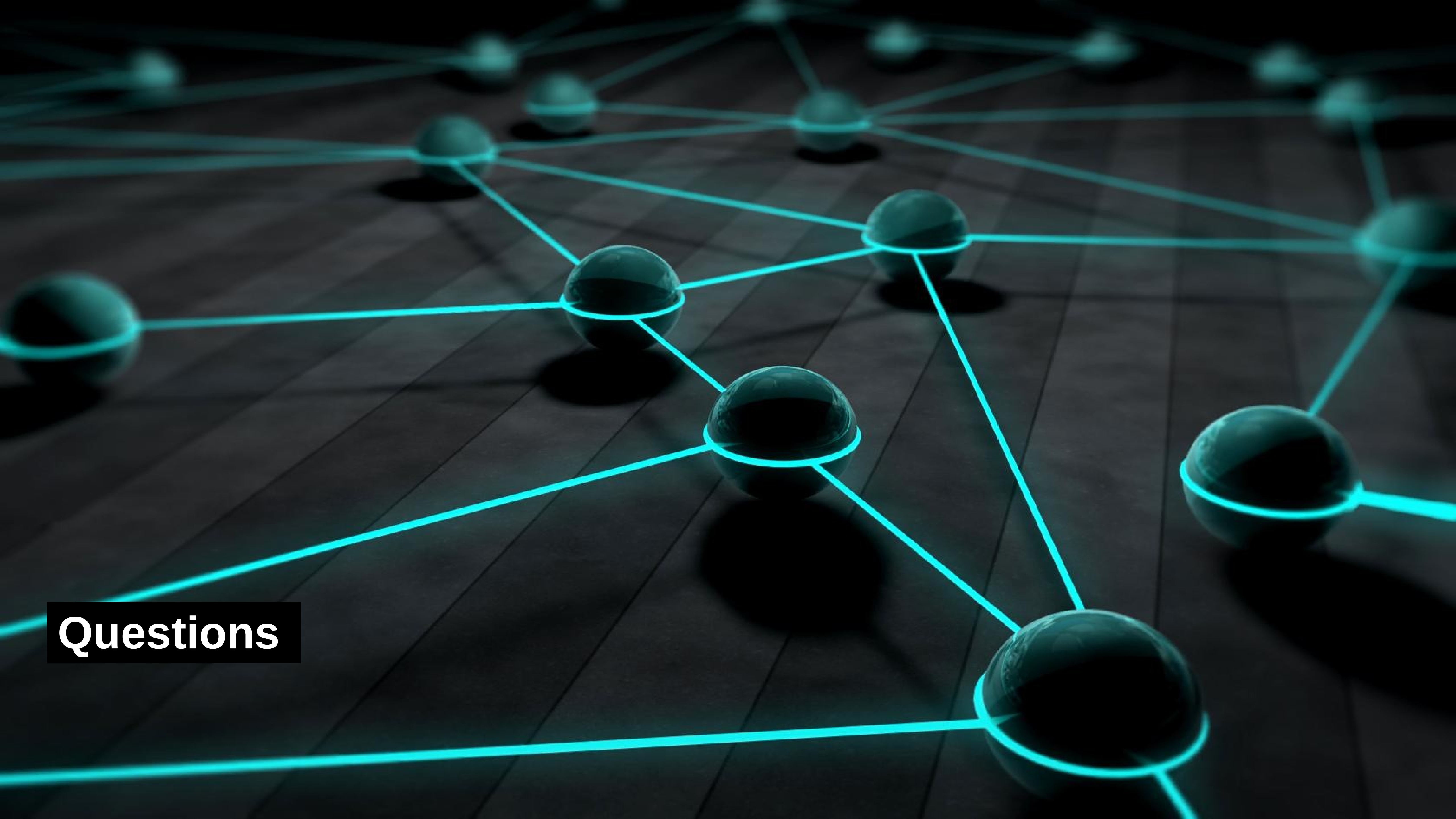
Priorities and Levels

Level	Priorities	Possible Interpretation
L1	12, 18, 27	High severity, likely, inexpensive to repair
L2	6, 8, 9	Medium severity, probable, medium cost to repair
L3	1, 2, 3, 4	Low severity, unlikely, expensive to repair



CERT Compliance Levels

- Baseline Compliance
  - All rules are followed.
  - Minimum bar for calling code “CERT-compliant.”
- Strong Compliance
  - All rules + majority of recommendations implemented.
  - Reduces risk of subtle, less obvious flaws.
- Mature Compliance
  - Rules + recommendations + considerations consistently applied.
  - Represents an organization that treats secure coding as part of its engineering culture.


Questions