Secure Development

DEF (*CVE Format and Meaning*) Common Vulnerabilities and Exposures (CVE) is a list of publicly disclosed cybersecurity vulnerabilities and exposures. Each CVE entry includes:

- CVE ID: a unique identifier for the vulnerability.
- Description: a brief summary of the vulnerability.
- References: links to advisories or reports that provide more details.

DEF (*CVSS - Common Vulnerability Scoring System*) The Common Vulnerability Scoring System (CVSS) is a standard for assessing the severity of software vulnerabilities. It provides a numerical score based on the following metrics:

- **Base Metrics**: Characteristics of the vulnerability itself.
- **Temporal Metrics**: Characteristics that change over time (e.g., exploit availability).
- **Environmental Metrics**: Characteristics specific to the organization's environment.

The CVSS score helps organizations prioritize and respond to vulnerabilities based on their severity.

DEF (*CVSS Base Metrics*) The CVSS Base Metrics include:

- **Attack Vector**: How the vulnerability is exploited *(Local, Adjacent, Network)*.
- **Attack Complexity**: The level of expertise required to exploit the vulnerability *(Low, High)*.
- **Privileges Required**: The level of privileges an attacker needs to exploit the vulnerability *(None, Low, High)*.
- **User Interaction**: Whether user interaction is required to exploit the vulnerability *(None, Required)*.
- **Scope**: Whether the vulnerability impacts the vulnerable component only or can affect other components *(Unchanged, Changed)*.
- **Confidentiality, Integrity, Availability Impact**: The impact on the security properties of confidentiality, integrity, and availability *(None, Low, High)*.

DEF (*Common Weakness Enumeration (CWE)*) The CWE is a community-developed list of common software security weaknesses. It provides a classification of software vulnerabilities to help developers understand and mitigate risks.

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, along with the corresponding security properties that each threat type attacks:

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

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."

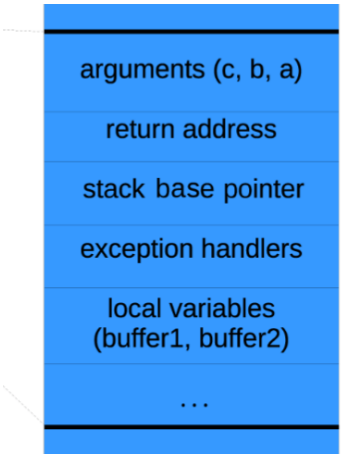
This highlights the importance of comprehensive testing strategies and the understanding that passing tests do not guarantee that no bugs exist in the software.

DEF (*Building Security In Maturity Model (BSIMM)*) It is a blueprint/software security framework for following good practices of software development. It is data-driven and so the practices are collated by examining the strategies that companies are utilising to produce secure software. The framework is composed of four domains: Governance, Intelligence, SSDL Touchpoints, and Deployment, with each domain containing three practices (for a total of 12 practices).

- Governance: Establishing security policies and practices.
  - Strategy and metrics
  - Compliance and policy.
  - Training.
- Intelligence: Gathering security metrics and threat intelligence.
  - Attack models.
  - Security features.
  - Standards and requirements.
- Development: Integrating security into the development process.
  - Code review.
  - Security testing.
  - Architecture analysis.
- Deployment: Ensuring secure deployment and operations.
  - Environment hardening.
  - Software environment.
  - Incident management.

Programming Errors

DEF (*Stack Layout*)



The above diagram shows the layout of the stack in memory. The stack grows downwards, with the stack pointer pointing to the top of the stack. The stack frame contains the return address, arguments, local variables, and saved registers. The frame pointer points to the base of the current stack frame (current function).

DEF (*Buffer Overflow*) Buffer overflows arise when a region of memory exceeds its allocated size, resulting in the overwriting of adjacent memory that may be used elsewhere in the program. This can lead to unexpected behavior in the program's execution. A buffer overflow exploits the stack layout to typically overwrite the return address of a function to perform arbitrary code execution (shellcode or return-to-libc). This attacks the integrity of the program and can lead to privilege escalation. It's also possible to attack the availability of the program by causing a denial of service - by crashing the program.

DEF (*Out-by-one/Arithmetic Errors*) Can lead to buffer overflows, memory corruption, and other security vulnerabilities.

DEF (*Faulty/Missing Error Condition Checking*) Not checking for error conditions can lead to unexpected behavior and security vulnerabilities.

DEF (*Format String Vulnerabilities*) Can lead to memory corruption, arbitrary code execution, and other security vulnerabilities. In C and C++, the `printf` function is vulnerable to format string attacks if the format string is user-controlled.

Platform and Operating System Level Attacks

DEF (*Assembly Programs and Shell-Code Attacks*) Assembly programs can be used to create shellcode which is injected into vulnerable applications through buffer overflows. Could spawn a shell (arbitrary code execution), reverse shell (remote code execution), or other malicious activities.

DEF (*OS Command Injection/Environment Variables*) It's important to be careful with programs that use environment variables to execute commands. In particular `PATH` and `LD_LIBRARY_PATH` can be manipulated to execute arbitrary code.

DEF (*Unix File Permission Codes*) Unix file permissions are represented by a 10 character string. The first character is the file type, the next three are the owner's permissions, the next three are the group's permissions, and the last three are the world's permissions (everyone else). The permissions can be:

- `r` - read (4)
- `w` - write (2)
- `x` - execute (1)

Common file types are:

- `-` - regular file
- `d` - directory
- `l` - symbolic link

DEF (*Permission Attacks*) Exploring the file system and looking for files with weak permissions can be a good way to find vulnerabilities.

DEF (*Stack Canary / StackGuard*) A form of **Tamper Detection** that can be used to detect buffer overflows. StackGuard canaries prevent buffer overflow attacks by inserting random values near the return pointer and checking their integrity before returning. While effective against return pointer overwriting, they don't protect against modifying local variables or preventing Denial of Service attacks that intentionally trigger canary detection. Also not effective against return-to-libc (ret2libc) or return-oriented programming (ROP) attacks.
DEF (*Control-Flow Integrity (CFI)*) A more powerful defense mechanism that ensures the code execution follows a pre-defined control graph. This *could* prevent return-to-libc and ROP attacks.

DEF (*Seperation Mechanisms*)

- Hardware Memory Protection
  - Memory Fences: These seperate memory accesses between OS and user code, providing one-way protection
  - Base and Bounds Registers: Enforce seperation between programs by controlling access to specific memory ranges, thus preventing unauthorized memory access
  - Tagged Architecture: Assign tags to memory locations that dictate access rights (read,write,execute), although not widely used in modern systems
- Paging splits programs/data into pieces.
  - Data Execution Prevention (DEP) / Write XOR Execute (WˆX): Prevents code from executing in data regions (making injected shellcode non-executable)
- Process Isolation (EXTRA)
  - Virtual Memory: Each process operates in its own virtual address space, preventing unauthorized access to other processes
  - Containerisation: Isolates processes in containers, preventing unauthorized access to other processes. Reduces the attack surface area and worst-case it will only compromise the container

DEF (*Diversification*) Make many versions of same program; thwarting attacks that rely on knowing the exact layout of the program in memory. **Address Space Layout Randomization (ASLR)** is a common technique that randomizes the memory layout of a program. Could still be suspectible to a brute-force attack of guessing the memory layout.

SQL Injection

SQL Injection

DEF (*SQL Injection*) SQL injection is a code injection technique that exploits vulnerabilities in an application's software by inserting malicious SQL statements into input fields.

Web Application Attacks and Defenses

Other Application Attacks

Advanced Secure Programming

Software Protection

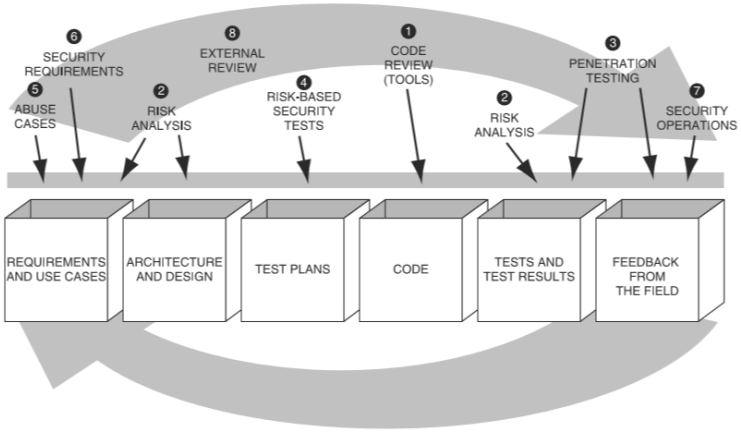


Figure 1: McGraw's Secure Software Development Lifecycle (SSDLC) emphasizes integrating security at various stages of software development.