CWE/SANS Top 10 Most Dangerous Software Errors

1. CWE-787: Out-of-Bounds Write
   This error occurs when data is written outside the allocated memory boundaries, potentially overwriting adjacent memory. It can lead to crashes, data corruption, or code execution. For example, buffer overflows in C applications may allow attackers to inject malicious code. Defensive programming includes bounds checking before writing to memory, using safe functions like strncpy instead of strcpy, and implementing memory-safe languages where possible.

2. CWE-79: Cross-Site Scripting (XSS)
   XSS vulnerabilities arise when user inputs are improperly neutralised, allowing attackers to inject scripts into web pages viewed by others. This can steal cookies, session tokens, or redirect users. Mitigate risks by encoding outputs, validating and sanitising inputs, and using security headers like Content Security Policy (CSP).

3. CWE-89: SQL Injection
   SQL injection occurs when untrusted data is embedded in SQL queries without proper sanitisation, allowing attackers to manipulate databases. For instance, SELECT * FROM users WHERE id='1 OR 1=1' could grant unauthorised access. Use prepared statements with parameterised queries and ORM frameworks to prevent this.

4. CWE-416: Use After Free
   This error happens when a program continues to use memory after it has been freed, leading to crashes or arbitrary code execution. Common in C/C++, it can be mitigated by nullifying pointers after freeing memory and employing memory-safe languages or tools like AddressSanitizer.

5. CWE-78: OS Command Injection
   Occurs when user inputs are improperly sanitised, enabling attackers to execute arbitrary OS commands. For example, system("ping " + user_input) could be exploited. Prevent this by avoiding command execution with user inputs, using APIs instead, and validating/sanitising inputs strictly.

6. CWE-20: Improper Input Validation
   Failure to validate inputs can cause security flaws like buffer overflows or injection attacks. For example, unchecked file uploads may allow malicious files. Apply strict validation (whitelisting preferred), enforce data type checks, and limit input lengths.

7. CWE-125: Out-of-Bounds Read
   Occurs when reading outside the memory bounds, potentially exposing sensitive data. This can happen in C programs due to faulty array indexing. Defend by implementing bounds checks, using memory-safe functions, and conducting rigorous testing.

8. CWE-22: Path Traversal
   Path traversal exploits improper restrictions on file path inputs, allowing access to unintended directories (e.g., ../../etc/passwd). Prevent by sanitising inputs, using secure APIs for file access, and restricting application permissions.

9. CWE-352: Cross-Site Request Forgery (CSRF)
   CSRF tricks users into executing unwanted actions in a web app where they're authenticated. For example, a hidden form submission could transfer funds. Mitigate by using anti-CSRF tokens, same-site cookies, and requiring re-authentication for sensitive actions.

10. CWE-434: Unrestricted File Upload
    This flaw allows attackers to upload files with dangerous types, potentially executing malicious code. For instance, uploading a PHP file to a web server. Defend by restricting allowed file types, scanning files for malware, and storing uploads outside the web root with secure permissions.

Secure Coding Practices

Validate Input

Ensuring that all input from untrusted sources is validated prevents malicious data from causing harm. For instance, validating user input in web forms can prevent SQL injection attacks. By checking that inputs conform to expected formats and ranges, applications can avoid processing unexpected or harmful data.

Heed Compiler Warnings

Compiling code with the highest warning levels and addressing all warnings can reveal potential vulnerabilities. For example, a compiler warning about a possible buffer overflow should be investigated and corrected to prevent security issues. Utilising static and dynamic analysis tools further aids in identifying and eliminating security flaws.

Architect and Design for Security Policies

Incorporating security policies during the design phase ensures that the software enforces necessary protections. For example, designing an application with separate modules that operate with the minimum required privileges reduces the risk of a security breach. This approach ensures that security is a fundamental aspect of the system's architecture.

Keep It Simple

Maintaining simplicity in design reduces the likelihood of errors that could lead to vulnerabilities. Complex systems are harder to secure and more prone to misconfigurations. For instance, using straightforward authentication mechanisms instead of intricate custom protocols can minimise security risks.

Default Deny

Implementing a default deny policy means that access is denied unless explicitly allowed. For example, a firewall configured to block all traffic except for specific permitted services reduces the attack surface. This principle ensures that only necessary and authorised actions are permitted.

Adhere to the Principle of Least Privilege

Granting processes and users only the minimum privileges necessary limits potential damage from security breaches. For instance, running a web server with non-administrative privileges prevents attackers from gaining full system control if the server is compromised.

Sanitise Data Sent to Other Systems

Cleaning and validating data before sending it to external systems prevents injection attacks. For example, sanitising inputs before including them in SQL queries prevents SQL injection vulnerabilities. This practice ensures that data cannot be used to exploit vulnerabilities in other systems.

Practice Defence in Depth

Employing multiple layers of security controls ensures that if one layer fails, others still provide protection. For example, combining network firewalls, intrusion detection systems, and secure coding practices creates a robust defence against attacks.

Use Effective Quality Assurance Techniques

Incorporating practices like fuzz testing, penetration testing, and code reviews helps identify and eliminate vulnerabilities. For instance, fuzz testing can uncover how software handles unexpected or random inputs, revealing potential security flaws.

Adopt a Secure Coding Standard

Following established secure coding standards ensures consistency and comprehensiveness in addressing security concerns. For example, adhering to the CERT C Coding Standard provides guidelines to prevent common vulnerabilities in C programs.