Secure Coding

Secure Coding

- As the size and complexity of a system increases, ensuring whether it has no vulnerability becomes difficult
- Generally, it is much less expensive to build secure software than to correct security issues after the software package has been completed, not to mention the costs that may be associated with a security breach.
- For better security, programmers should follow certain guidelines while coding
- OWASP has created a set of general software security coding practices, in a checklist format, that can be integrated into the software development lifecycle. Implementation of these practices will mitigate most common software vulnerabilities.

Selecting Programming Language

- For better security, a programming language should provide
  - Memory safety: automatically manage memory and prevent common vulnerabilities like buffer overflows and dangling pointers.

    Languages that are memory safe: Rust, Go, C#, Java, Swift, Python, and JavaScript

    Languages that are not memory safe: C, C++
  - Type safety:It is an abstract construct that enables the language to avoid type errors
  - Sandboxing: Running untrusted or risky code within a secure, isolated environment to prevent it from causing harm to the host system or accessing unauthorized resources
  - Security framework/API: provide functionalities to developers for implementing secure applications and protecting sensitive data. Example: Input validation, cryptography

Secure Input

- Validation
  - Checking characteristics of the input such as size, its domain
  - The validation has to be done before data are used in the application
  - It can be done using regular expressions, standard libraries, etc.,

- Canonicalisation
  - Transforming the values of input data into a standard form
  - This operation is necessary for values that can be represented in alternative ways, such as absolute and relative file paths

Secure Input

- Sanitisation
  - Transforming the values of input data by removing some of their parts or writing them differently. Example: javascript
  - Must be performed before using the input data
  - Typically done by applying several filters on the values of the input data, by escaping special characters, or by using prepared statements when working with DBMS.

Input Validation

- Validating the input is crucial for correctness, security and safety.
- Not validating the input may lead to several security issues
- **Buffer overflow**
- **Integer Overflow:** When the result of an arithmetic operation, such as multiplication or addition, exceeds the maximum size of the integer type used to store it.
    - An integer overflow during a buffer length calculation can result in allocating a buffer that is too small to hold the data to be copied into it. A buffer overflow can result when the data is copied.
    - When calculating a purchase order total, an integer overflow could allow the total to shift from a positive value to a negative one. This would, in effect, give money to the customer in addition to their purchases, when the transaction is completed.

Input Validation

- Choose an integer type used for a variable that is consistent with the functions to be performed.
- The operands of an integer operation and/or the result of it should be checked for overflow conditions.
  - if A and B are both unsigned integers, then we can check A + B < A
  - We can also check before the operation: check if B > SIZE_MAX - A
  - These checks can become very complicated when integers of different sign, size, and order of operations are considered.
  - Use safe integer libraries, such as "SafeInt"

Secure Coding Checklist

- Input Validation
- Output Encoding
- Authentication and Password Management
- Session Management
- Access Control
- Cryptographic Practices
- Error Handling and Logging
- Data Protection
- Communication Security
- System Configuration
- Database Security
- File Management
- Memory Management
- General Coding Practices

Secure Coding Checklist

- Authentication and Password Management
- Session Management
- Access Control
- **Cryptographic Practices :** Securing master key, random number generation
- **Error Handling and Logging :** Log all crucial activities (authentication attempts, access control failures, etc), Do not log sensitive information,
- **Data Protection:** Protect sensitive data (such as passwords) using encryption, access control, Use principle of least privilege
- **Communication Security:** Securing data at all layers

Secure Coding Checklist

- **System Configuration:** use secure versions/models of softwares and hardwares, regularly apply patches, remove/disable unnecessary components and functionalities
- **Database Security:** use strongly typed parameterized queries, use strong authentication, remove/disable unnecessary users and functionalities, change all default passwords
- **File management:** Validate uploaded files, access control, authenticate before allowing uploading, control file types and execution
- **Memory management:** Validate input and output, Boundary check, Use secure functions, secure release of acquired memory

Code Analysis

- Static Analysis: Analyses the code without executing it.
  Also referred to as Static Application Security Testing (SAST)
- Dynamic Analysis: Analyses the code at run-time.
  Also referred to as Dynamic Application Security Testing (DAST)

Static Application Security Testing (SAST)

- Generally requires access to source code
- Scans for all theoretical paths/vulnerabilities
- May generate more false positives
- Can be done as part of code review
- Examples:
  - Polyspace Code Prover: static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and other run-time errors in C and C++ source code.
  - Clang Static Analyzer: a static analyzer which can be used to find bugs using path sensitive analysis. Used for C, C++, and Objective-C.

Dynamic Application Security Testing

- Doesn’t require access to source code
- Static analysis doesn't catch every code defect.
- Can be done as part of testing.
- Generates relatively less false positives
- Examples of DAST tools used for Web Applications
    - Invicti
    - Acunetix

Common Weakness Enumeration (CWE)

- A community-developed catalog that features hardware and software weaknesses
- Managed by MITRE
- Described as “a common language, a measuring stick for security tools, and a baseline for weakness identification, mitigation, and prevention efforts.”
- Has a **top 25** list. This list demonstrates the currently most common and impactful software weaknesses.
- Often easy to find and exploit, these can lead to exploitable vulnerabilities that allow adversaries to completely take over a system, steal data, or prevent applications from working.

Common Weakness Enumeration

- The CWE list prioritizes weaknesses. The top 25 entries are prioritized using input from more than two dozen different organizations.
- They evaluate each weakness based on frequency and importance. Many of the weaknesses (in C programs) listed in CWE relate to buffer overflow.

References

1. https://www.memorysafety.org/docs/memory-safety/
2. https://medium.com/codex/three-operations-on-input-data-to-make-your-software-more-secure-e5fc5aca2e70
3. https://blog.hubspot.com/website/what-is-utf-8
4. https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/02-checklist/05-checklist
5. https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/assets/docs/OWASP_SCP_Quick_Reference_Guide_v21.pdf
6. http://projects.webappsec.org/w/page/13246946/Integer%20Overflows
7. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html
8. https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards