Executable Security Standards (ESS)

Language Agnostic Security Specific Rules

October 20, 2024

JOBY REUBEN
Auguth Research Foundation
Bangalore, India
joby@auguth.org

ABSTRACT

Executable Security Standard (ESS) defines a mandatory set of standards that has to be followed while programming executables written in different high level languages to ensure utmost security and maintenance practices. The ESS provides a language-agnostic set of standards that should be in compliance to assure intended safe execution. The primary goal of ESS is to provide safety over computation. This set of rules can be utilized for writing safe executable programs that can be deployed to ALAN. The ESS has derived rules from some of the language-specific security standards such as *MISRA & CERT* standards and provides a comprehensive language-agnostic security rules.

Contents

A. Codebase

1) *Codebase Definition*: A codebase is a separate code file or an application-specific source folder, including code files, test cases, and related resources.
2) *Function Categorization*: Functions must be organized by category or operation, within separate codebases for different operations (e.g., encoding, state, runtime, etc)
3) *Data Type Codebase*: New data type definitions or type alias used throughout multiple codebases must be declared in a seperate codebase.
4) *Loose Coupling*: Codebases must be loosely coupled, interacting only through its defined interface function i.e., public functions.
5) *Interface Functions*: Each loosely coupled module or codebase must have interface functions which may call its internal private functions.
6) *Function Cohesion*: Functions within a codebase must be highly cohesive, performing a single, definitive operation.
7) *Utility Separation*: Utility functions or application-agnostic functions must be separated from application-specific codebases.
8) *Inter-Module Interaction*: Functions from one loosely coupled module or codebase must not directly use functions from another, except for utility or application-agnostic functions.
9) *Eliminate Unreachable Code*: Avoid including unreachable code that is never executed due to control flow reasons. Such code should be identified and removed.
10) *Remove Dead Code*: Do not maintain dead code—code that is written but never executed, regardless of execution paths or conditions. Eliminate redundant, obsolete, or unnecessary code.
11) *Avoid Unused Declarations*: Ensure that the codebase does not contain unused type declarations, such as structs, enums, type aliases or any declarations that are defined but never used or referenced.
12) *Distinct and Unique External Identifiers*: External identifiers, including function names, global variables, and type aliases, should be distinct and unique across all codebases.
13) *Unique Internal Identifiers*: Internal identifiers within a codebase or within a function should be unique to avoid conflicts. Proper names should be given according to the definitive purpose they serve.
14) *Global Variables and Definitions*: Global variables, definitions, and functions should only be defined in a single codebase and should not be duplicated across different codebases. Categorical codebases should be maintained for global definitions serving distinct purposes.
15) *Internal Definitions*: Definitions specific to a codebase should not be used externally by other codebases. Only global definitions, which are defined in global specified codebases, should be utilized externally.
16) *Adopt Structured Programming*: Use structured programming practices for codebases. Avoid using `goto` statements or labels to jump to different logic segments, ensuring a linear flow.
17) *Avoid Commenting Out Code*: Use version control systems to access previous versions of the source code; commenting out sections of code is strictly prohibited for preventing accidental reintroduction.
18) *Clean File Names*: Source code file names should be clean and concise, avoiding symbols (except for underscores as space substitutes) and spaces to prevent compiler and file system incompatibility.
19) *Distinct Identifiers* : In a loosely coupled codebase, identifiers names (e.g., variable, function, type aliases) must be distinct and clearly reflect their single, well-defined purpose or opera-

tion.

20) *Avoid Ambiguity*: Identifiers with similar names that differ only by case or symbols are strictly prohibited. Names should be explicit and unambiguous, clearly indicating their purpose within the codebase.
21) *Comments Not for Dead-Code*: Comments should contain only general information for maintainability and clarity, and must not include dead code.
22) *Avoiding Visually Similar Characters*: Avoid typographically similar identifier names, e.g., do not use identifier characters that resemble one another, such as I and l.
23) *One-Definition Rule*: Each entity in a codebase (such as a function, variable, or type) must have exactly one definition within the entire codebase/application.
24) *Avoid Global Identifiers Conflicts*: Any identifier used in multiple codebases should be uniquely defined and must not conflict with identifiers in other codebases of the application.
25) *Import Declarations at Utilization Area*: External linkage of other codebases and their identifiers should be declared at the top of the utilization line and included in a comment for clarity and maintenance purposes.
26) *Identifiers with Specializations*: All specializations of identifiers should be included in the same codebase as their primary declarations.
27) *Comments Structure Rule*: Comments should precede the code they describe and must not be placed on the same line as the code. Avoid excessive whitespace within comments.
28) *Unique Module Names*: Ensure internal and external module or library names are unique.
29) *Clarifying Relationships*: Clarify the relationship between two logical statements/expressions/declarations in comments or via explicit identifier naming.
30) *Initialized Variables Only*: Use initialized or defined variables; never rely on execution involving undefined identifiers.
31) *Module data access restrictions*: Data from one module/codebase should not be accessed by another module/codebase unless explicit access to bounded data is provided for specific operations.
32) *Remove debugging entrypoints in production*: Production code should not contain any debugging entry points.
33) *Confinement of security primitives*: Ensure that code utilizing security primitives is confined to an individual codebase.
34) *Permission for Cross-Calling*: Specify permission for cross-calling between modules/functions; do not respond if the requesting module/function lacks permission.

B. Types

1) *Create Explicit Type Aliases*: If languages offer type definitions with names that doesn't specify explicit bounds or signedness, create type aliases that clearly specify memory bounds (e.g., 8-bit) and signedness (e.g., unsigned for non-negative values, signed for both positive and negative values).
2) *Use Static Types*: Prefer using static types with known bounds that are checked at compile time.
3) *Favor Stack-Based Types*: Prioritize stack-based types over heap-based dynamic types and avoid issues associated with heap management.
4) *Prefer Immutable Data Types*: Use immutable data types whenever possible. Opt for mutable data types only when the logic cannot be defined immutably.
5) *Avoid Abstracted Data Types for Low-Level Values*: Avoid using types that result in inefficient memory usage. Instead, use appropriate low-level types for memory efficiency.
6) *Byte Sequences*: Use usinged 8-bit integers i.e., `u8` arrays i.e., `ary[u8:N]` or vectors i.e., `vec[u8]` for representing byte sequences.
7) *Type Specification*: All variables, definitions, function parameters (inputs and outputs) must specify their type or an enum variant. Avoid languages with dynamic types.
8) *Unique Enum Values*: Each implicitly-specified enum variant must have a unique value, always ensure there are no conflicts, even if the values are assigned by the compiler.
9) *Overflow and Wrapping*: Overflows should be predetermined, and wrapping should be explicitly specified if allowed. Ensure that no runtime panics occur by handling it appropriately.
10) *Avoid Unions*: Unions should be avoided in favor of strict, well-defined types.
11) *Prefer References*: Use references i.e., smart pointers instead of raw pointers to ensure safer memory access.
12) *Manage Conversion of Data Types*: Carefully manage the conversion i.e., casting of data types with different memory layouts.
13) *Function Qualifiers*: Function types should not use qualifiers like mutable, immutable, constant, concurrent, or atomic, which are intended for modifying variables, not function types.
14) *Prefer Safe Data Types*: Always use sub-sequence data types such as slices and avoid variable or flexible array types, which are intended to be static during compile time. Instead, use dynamic arrays, such as vectors, which may allocate their data on the heap.
15) *Use Fixed Integer Sizes*: Always use fixed integer sizes, such as `u32`/`i32` or `u64`/`i64`, for size expression's return values i.e., `u-size` or `i-size`. To avoid architecture-specific issues, either fix an executable version for a specific architecture or create versions of executables for the desired architecture for performance effectiveness.
16) *Implement Error Handling*: Always use special error handling types, such as `Option`, or ensure that there is a null variant for functions or expressions that may encounter errors or inconsistencies.
17) *Array Size Declaration*: Array sizes should be defined as constants, avoiding the representation of a minimum number of elements.
18) *Never Use Intermediate Variables*: Avoid intermediate/temporary or logically duplicated variables that possess the same type qualifier and are used only once.
19) *Escape Sequences Rule*: Avoid escape sequences in strings that are not defined by the language.
20) *Pre & Suffix in Asssigned Values*: Avoid type value suffixes or prefixes; instead, use new type aliases that define the type's bounds and signedness.
21) *Avoid Unsafe Macros*: Refrain from using macros that rely on textual substitutions without inherent type safety. Prefer languages that ensure safety by limiting textual substitution models and tightly integrating macro syntax with the language syntax.
22) *Avoid Unsafe Preproccessor Directives*: Refrain from using arbitrary textual substitutions such as preproccessor directives for conditional compilation.
23) *Variable Instances Consistency*: Every instance of an object used across the codebase must be consistent with its declared data type.
24) *Avoid Bit-Value Assignment*: Always prefer safe-types to declare values, rather than directly assigning values at the bit level.
25) *Value 0 is not Null*: The integer/literal/value zero (`0`) should not

be treated as a `null` type. Use a specific `null` type provided by the language instead.

26) *Null as Enum Variant*: `null` can be considered an enum variant, as applicable in the Option Type `(null, some)`. All enum variant rules apply to `null`.
27) *Choosing Types for its Nature*: Use types for their intended purpose; for example, do not use char type to store integer values, as it is strictly for storing characters such as UTF-8 characters.
28) *Types for Exception Handling*: Use `Result` and `Option` types to simplify exception and handling.
29) *Conversion of Base to Derived*: Converting or casting a base type/identifier to a derived type/identifier should not occur on hierarchical structures.
30) *Never use Raw-Pointers*: Do not use raw pointer casting i.e., conversion; avoid pointer-based expressions, operations, and initiations.
31) *Qualifiers during Casting*: When casting, any qualifiers such as mutable, immutable, atomic, constants, or references should not be ignored, removed, or unwrapped from the new casted type.
32) *Never Cast Function Types*: Function types which are not similar to variables should not be cast to any new data type.
33) *Floating Point Integers Precision-Issues*: Floating-point types should avoid using equality and inequality operators due to varying precision issues; instead, use a range-based approach to check equality within a specific threshold.
34) *Braces on Array Initialization*: Braces must be used for non-zero initialization of arrays or similar data types.
35) *Enum Variant Consistent Initialization*: Enumerator lists should utilize a consistent method of initialization for all its variants.
36) *Bit-Fields Types*: In cases of using bit-fields for compact representation, restrict the type to boolean or a signed or unsigned type based on the nature of the bit field.
37) *Sign Bit on Bit-Field Types*: Bit-fields that require signedness should ensure that the number of bits the type occupies is greater than two to explicitly accommodate the sign bit.
38) *Abstract Data Types*: While using abstracted data types, avoid revealing the underlying structure to the function; only necessitate its intended logical execution on the abstracted data type and expose the minimum amount of objects.
39) *Avoid Floating-Point Issues*: Avoid using floating-point integer types with rounding, arithmetic, and precision issues in critical code run by various hardware.
40) *No Denormalized Floating Points*: Do not use denormalized floating-point numbers as values.
41) *Integer for Precise Computation*: For precise computation, use integer data types instead of floating point types.
42) *Fixed Array Sizes*: Do not use types that provide variable length arrays; the array size should be fixed during compile time.
43) *Array Bounds Specification*: Specify array bounds, i.e., size in every instance of initialization or when using the initialized array variable.
44) *String Literal Consistency*: Avoid concatenating different types of string literals; instead, use consistent types.
45) *Returning Empty Arrays*: Prefer returning an empty array over a `null` value, providing a flag that the array has zero elements.
46) *Compatible Value Types*: Compatible values should have the same type; avoid type mismatch and loss of precision.
47) *Standard Layout Types*: Always avoid non-standard layout types; prefer POD and standard layout types.
48) *Internal Representation*: Avoid manipulating any internal or hidden representation of a defined object.
49) *Moved Objects*: Do not depend on a moved object's value.
50) *Complete and valid encoded data*: Encoded data types such as character or string should be complete and valid; avoid partial or invalid data.
51) *Use Option type*: Use `Option` type instead of directly relying on null for absence of value.
52) *Handle exceptional floating point values*: Handle exceptional floating point values such as NaN (Not a Number), positive and negative infinity.
53) *Correct underlying data types*: Only operate or inspect values using their correct underlying data types rather than relying on their string representation or any abstracted forms.
54) *Mutable Reference Types*: Never define or declare a mutable reference type, i.e., a mutable smart pointer.
55) *Avoid references to mutable data types*: Do not create references to mutable data types.
56) *Enumeration Labels*: Do not rely on numeric values associated with enumeration variants; treat them only as labels.
57) *Immutable Numeric Constants*: Numeric constants with immutable qualifiers must not be updated in later releases.
58) *Avoid Generics for High Security*: Do not use generics where security requirements are high.
59) *Use Option types for null safety*: Use `Option` types to safely handle null variants of data types and avoid null pointer exceptions.
60) *Standard encoding for strings and characters*: Ensure that string and character values should be encoded from standard formats.
61) *Valid scope for data references*: References to data should only be initialized within the valid scope of that data.
62) *Enumerations for Short Lists*: Use enumerations for short lists of variants; for longer lists, prefer collection data types.

C. Declarations

1) *Initialize Definitions*: Definitions must be initialized before being read or operated on. Prefer safe initializations over assigning empty values, especially for sequences, structs, and complex data structures.
2) *Avoid Redundant Initialization*: Avoid redundant modification or initialization of variables. Ensure that each variable is initialized only once.
3) *Avoid Identifier Shadowing*: Identifiers declared in an inner scope should not hide or shadow identifiers declared in an outer scope.
4) *Avoid Undefined Constructs*: Avoid definitions that negate a specific expression or logic. Instead, define constructs that are guaranteed to be used within the codebase. To remove a construct, eliminate its definition from the codebase entirely.
5) *Eliminate Dormant Objects*: All variables and definitions declared, regardless of their qualifier, must be utilized in expressions. Dormant objects should not exist in the codebase.
6) *Data Type Modifiability*: Define data types only to be modifiable for their intended operation and purpose, and not globally modifiable.
7) *Identifiers Inside Scopes*: Identifiers declared in inner scopes should not have the same name as those in outer scopes.
8) *Never Re-use Identifiers*: An identifier utilized must not be reused for an object or function within the same scope.
9) *Global Scope Usage*: Maintain clarity with identifiers used within their scope and minimize global scope usage.
10) *Specializations Variants*: Identifiers/objects should either have specializations defined for all variants or none at all (e.g.,

`identifier_i8`, `identifier_u8`, or `identifier`) to avoid unintended errors during function calls.

11) *Conflict-Free Identifiers*: Identifiers should not declare multiple qualifiers that conflict with each other; prefer conflict-free qualifier declaration, or at most, a single qualifier.
12) *Reserved Identifier Conflicts*: Never define new identifiers that have naming conflicts with language or library reserved identifiers.
13) *Derived Identifier Name Resolution*: A declared identifier within a hierarchical structure must be accessed by including all base identifiers to ensure clear name resolution.
14) *Initialization Order*: Have a well-defined initialization order of identifiers and objects; avoid circular dependencies.
15) *Prefer bounded initializations*: Prefer bounded initializations throughout all constructs defined in the codebase.
16) *Avoid nested wrappers*: Do not create nested wrappers for individual types or resources.
17) *Variable Scope*: If a variable is used only within a single function, it should be declared at the block scope (inside the function) rather than at the file scope and vice versa.

D. Expressions

1) *Promote Cohesion in Expressions*: To minimize side effects, expressions should be highly cohesive and broken down into finer, more focused expressions.
2) *Avoid Raw Pointers*: Refrain from using raw pointers or their equivalent pointer related expressions, including arithmetic operations, comparisons, and conversions, under any circumstances.
3) *Correct Placement of Statements*: Statements, expressions, and declarations should be correctly placed in accordance with the intended preceding and succeeding logical statements or expressions as dictated by the language's syntax.
4) *No Side Effects*: Initialization logic or expressions should be free of side effects beyond their immediate scope, such as modifying global variables, performing I/O operations, or altering the state of external systems.
5) *Use Parentheses for Clarity*: Use parentheses to make the precedence of operators explicit within expressions. This enhances clarity and prevents unintended behavior. For example, in the expression $3 + 4 * 5$, multiplication has higher precedence than addition, so the result is $3 + (4 * 5)$.
6) *Write Expressions Separately*: Expressions should be written on separate lines rather than being separated by commas within the same line.
7) *Only Define Logically-Executable Expressions*: Avoid expressions that are logically impossible to execute; ensure that all possible execution paths are valid and reachable under some conditions.
8) *Avoid Sub-Expressions*: Expressions should be highly cohesive; sub-expressions should be avoided.

E. Functions

1) *Parameter Naming and Documentation*: Name function parameters according to their purpose.
2) *Pupose of Function Parameters*: Use comments to explain the purpose of function parameter names if it isn't self-explanatory.
3) *Ensure Strict Typing*: Functions must strictly enforce types for all variables to prevent type-related errors and ensure correct behavior.
4) *Global Function Rule*: The identifier `main` shall not be used for any function other than the global function `main`.
5) *Function Access Control*: Define functions as public if they need to be accessed by other codebases i.e., interface functions in loosely coupled codebases.
6) *Cohesive Functions*: Define highly cohesive functions under interface functions as private since they should only be accessible by the interface functions.
7) *Utility Functions*: General utility functions (e.g., bit and byte manipulation) can be declared public and defined in a separate codebase.
8) *Function Re-Declarations Rule*: Avoid function re-declarations; instead, define functions in a single codebase for clarity.
9) *Avoid Unused Input Parameters*: Do not include unused input parameters in functions.
10) *Function Inlining*: Based on the compiled target purpose and expected executable size, determine whether to inline highly cohesive functions that are rigorously called in multiple instances.
11) *Inlining Interface Functions*: Interface functions i.e., public functions of loosely coupled codebases are exempt from inlining.
12) *Size/Length Functions*: Variable-length types or arrays shall include a size or length function in standard libraries to fetch the number of elements. When passing such a variable to a function, ensure that it is not mistakenly interpreted as the size of the pointer rather than the original array.
13) *Mandatory Return Values*: All functions with non-void return types must return a value consistent with their declared data type.
14) *Array Size in Function Parameters*: If a function parameter includes an array, the size/length identifier must also be specified in the function signature.
15) *Return Types*: Return types of functions should be proper, justifiable, and memory efficient. They must avoid inappropriate data types, including those perceived as raw binary values.
16) *Utilization of Function Outputs*: All returned outputs from functions must be utilized within the application or passed to other functions.
17) *Immutable Function Parameters*: Function parameters i.e., input and output arguments must be immutable, preventing any mutability of parameters inside & outside the function scope.
18) *No-Return Functions*: Functions designed to terminate control flow such as `thorw`, `exit`, distinct from other void functions, should use the proper keyword based on language syntax to indicate they do not return any value. Such functions should have a void return type.
19) *Function Calls*: Provide input parameters in the correct order i.e., index, during function calls to ensure proper execution and expected results
20) *Use Safe Data Type Functions*: Use data types with safely abstracted internal/external functions or operators for comparison, copying, and concatenation.
21) *Pointers Included Functions*: Avoid using functions that operate directly on pointers.
22) *Handle Type Conversion Functions*: Functions used for type conversions should be meticulously managed to detect invalid inputs and handle errors efficiently.
23) *Use Standardized Time Formats*: Avoid using time and date functions that may provide inaccurate or imprecise outputs. Instead, use standardized time formats, such as Unix Epoch Time.
24) *Access Control for APIs*: Ensure API or public functions are protected with strong access control or permissions.
25) *Validate Imported Functions*: Imported functions from external

libraries should handle invalid inputs, data inconsistencies, and overflow issues.

26) *Never Use Restricted Functions*: Functions with void or null return types that do not produce any state transitions should not exist in the codebase.
27) *Eliminate Dormant Functions*: All functions should have proper control flow and must be callable; dormant functions should not exist in the codebase.
28) *Arguments Utilization*: All arguments provided to a function must be utilized in a way that impacts the return value; if not, the argument must not be declared.
29) *Nested Function Declarations*: Functions should not be declared i.e., defined inside other functions; prefer externally calling functions declared at the file or class scope.
30) *Exceptions Handling of Cohesive Functions*: Error or exception handling must be specific to highly cohesive functions and must not be allowed outside the functions.
31) *Exception wrapped Parameters*: If parameters wrapped with exceptions are passed, the receiver must expect and handle the exception.
32) *Proper Function Signatures*: Functions utilized in multiple codebases should have similar signatures, including parameters and names.
33) *Explicit Single Return Rule*: All functions must include a single return statement to explicitly provide an exit path, even if the language does not require it.
34) *Utilization of Function-Identifiers*: A function identifier should only be used to either call the function directly or be used for a reference.
35) *Avoiding Mutable References*: Functions should not return references to mutable identifiers; ensure that returned references point to immutable objects/identifiers.
36) *Avoid Overloading Functions*: Do not use function overloading; instead, use a single function and prefer enums for undeterministic parameter types.
37) *Sanitized Data to Functions*: The data served to functions should only include sanitized data with bounds
38) *Explicit Function Definitions*: Define private and public functions explicitly to avoid importing private functions that should not be called by other codebases.
39) *Floating Point Casting*: Cast a floating-point return type of a function to an integer to avoid precision issues related to floating-point types.
40) *File Creation Return Value*: The return value of file creation functions should be ensured, as it may fail due to various reasons such as permission issues, system limitations, etc.
41) *Null-Terminated Strings*: Do not pass a non-null-terminated string to a function that expects a null-terminated string.
42) *Minimum Access Levels*: Grant the minimum level of access necessary for client-sde processes to perform their functions.
43) *Platform-Independent Functions*: Use platform-independent functions instead of relying on specific functions or libraries for a specific OS or kernel.
44) *Variadic Functions*: Variadic functions with a variable number of arguments should be avoided; prefer a static fixed number of arguments provided at compile time.
45) *Simple Signal Handlers*: Signal handler functions should only perform simple, reentrant operations that do not interact with shared resources.
46) *Return Value*: Any state changes that occur as a result of executing a function or expression must not extend beyond returning a value.
47) *Signal Handler Simplicity*: Signal handler functions should not change the handling of signals or modify the program's state in complex ways.
48) *No Return from Signal Handlers*: Signal handler functions must not return any value. Instead, they should perform necessary cleanup or logging and then terminate or recover the program as needed.
49) *Transient Failure Handling*: Wrap failing functions in a loop and handle transient failures are handled properly.
50) *Separate Success and Error Signals*: Do not mix success values with error indications in the same output. Use different ways to signal those outcomes.
51) *Reset Global Error Status*: If a program uses a global variable to register the last function call's error status, always reset this variable after reading it.
52) *Mask Non-Interruptible Signals*: Mask signals handled by non-interruptible signal handler functions to prevent them from being interrupted.
53) *Signals for Exceptional Conditions*: Avoid using signals or signal handler functions to implement normal program functionality; use them only for exceptional conditions or asynchronous events.
54) *Asynchronous-Safe Functions*: Call only asynchronous-safe functions within signal handler functions.
55) *Signal Handlers*: A signal handler function shall be a Plain Old function.
56) *Avoid exposing internal state*: Do not expose the internal state of a code base/modules from a public function; only use it for control flow to the module's private functions.
57) *Revert state on function failure*: On function failure, ensure that the program's state is reverted to its prior condition.
58) *Execute cleanup logic*: Ensure that cleanup logic is executed on a function's exit path regardless of errors handled.
59) *Do not define overridable functions*: Do not define or invoke overridable functions.
60) *Use standardized syntax for API functions*: Always define and use a strict syntax for safe invocation of API functions and avoid text-based or string-based submissions.
61) *Character Count Functions*: For determining the number of characters in a string, use suitable functions valid for the string data type's encoding format.

F. Operations

1) *Type Conversion*: Type conversion for logical operations should not narrow or lose precision. Keep return and input types consistent with the same nature.
2) *Valid Range for Shift Operations*: The value used to determine the number of positions to shift in a bit shift operation must be within the range from zero up to one less than the number of bits in the type being shifted.
3) *Avoid Operations in Conditional Evaluations*: The right-hand operand of logical `AND` ($\wedge$) or `OR` ($\vee$) operators should not include operations, as it may not be evaluated if the left-hand operand determines the condition.
4) *Ensure Signedness on Operands*: Operators use the same signedness for types. Simplify expressions into highly cohesive forms by including pre- and post-expressions that do not affect the signedness rule.
5) *Cast Types to Match during Operation*: When applying logical operators to types from other types, ensure both types are the same. If not, cast the types to match.
6) *Boolean restricted Operators*: Boolean types should be restricted to the following operators: Assignment, Logical `AND, OR, NOT`, Equal To, Not Equal To.

7) *Enum and Character restricted Operators*: Enum and character types should only use the following operators: Assignment, Equal To, Not Equal To.
8) *Avoiding Signed Types*: Never use signed types for bitwise operators.
9) *Cohesion on Operators*: The increment (++) and decrement (−−) operators should be cohesive.
10) *Avoid Overloading these Operators*: Do not overload the comma operator, `AND` operator, or `OR` operator.
11) *Ensure Boolean Representation*: Operands of operators such as `NOT`, `AND`, and `OR` should be of type boolean, or explicitly provide a boolean representation for correct evaluation.
12) *Signedness Effects on Operators*: Operators should be assigned to compatible types; for example, the minus operator (−) must not be applied to expressions with an unsigned type.
13) *Address-of-Operator for References*: The address-of operator should provide an abstracted reference instead of direct raw-pointer address access.
14) *Avoid Side-Effects of Certain Operands*: Operands of logical `AND` and `OR` operators should not have side effects.
15) *Avoiding Comma Operator*: The comma operator shall not be used in expressions where it returns the result of the last operand.
16) *Division & Modulo Safety*: Ensure that division and modulo operations do not include divide-by-zero expressions, resulting in undefined errors.
17) *Bitwise Shift Operations*: Never use bitwise shift operators with negative numbers and above the number of bits of the operand data type.
18) *Modulo Operator Usage*: Use the return data type as unsigned when using the modulo operator on unsigned operands, and signed operands if any one operand is signed.
19) *Bitwise Operands Restrictions*: Use bitwise operands on unsigned types or operands only.
20) *Sizeof Operator Usage*: Do not use the `sizeof` operator or attempt to return the size of a referenced array, as it may return the size of the reference instead of the underlying array data type.
21) *File Operations Clarity*: Operations after opening a file should be clear, i.e., `read, write, concat`; include only one method of operation with clear intent.
22) *String Truncation Prevention*: String operations should not truncate strings, leading to data loss or precision.
23) *Array Size Specification*: Operations or functions that utilize array data types must always specify the array size in all recurring expressions.
24) *Evaluated Operands*: Write operands that are surely evaluated; avoid unevaluated operands.
25) *Qualifier Constraints*: Expressions and operations should respect the qualifier constraints of the data type it utilizes.
26) *Iterators and Loop Counters*: Use iteration operations only on valid ranges of well-defined objects.
27) *Multiple Iterators*: Perform operations on multiple iterators only if they refer to the same data structure.
28) *Iterator Overflow*: Operations on iterators should not result in overflow; if so, it should be handled.
29) *Valid Access*: Use valid indices, iterators, or smart pointers (i.e., references) for operations that access elements of objects.
30) *Valid Range*: Pre-determine the valid range of elements for an object before executing operations or functions on them.
31) *Read/Write Correctness*: When switching between operations on the same object, such as reading and writing, ensure correctness.
32) *Character Encoding Consistency*: Keep character encoding consistent across codebases to ensure proper numeric value comparison or other operations.
33) *User-defined regional rules*: Do not ignore user-defined regional rules or format on values of data types that depend on them when performing operations.
34) *Operate on values of data types*: Operations should be performed on the values of wrapped or nested data types, rather than on their object or wrapped references.
35) *No simultaneous operations*: There should be no simultaneous multiple operations on an individual data item.
36) *Avoid NaN operations*: Avoid operations that involve NaN (Not a number) and ensure expressions do not result in NaN.
37) *Specify access permissions*: Specify access permissions for specific operations rather than granting general access to functions, operations or expressions that might use the data.
38) *No side effects during operations*: Ther should be no side effects on the operands during operation execution except for the intended operational effects.
39) *Hide sensitive information in errors*: Exceptions or error messages should not reveal any sensitive information.
40) *Prefer hexadecimal or raw binary format*: Prefer opting for hexadecimal or raw binary format for writing byte sequences in any operation.
41) *Specific operation permissions*: Granted permissions should be specific to an operation and are revoked once its execution is complete.
42) *Reference Equality Checks*: When using equality operators with references, ensure the operators check equality of the data they point to.

G. Loops and Conditionals

1) *Standard Loop Structure*: `for` loops must include initialization (`int i = 0`), condition ($i < 10$), iteration expression or loop counter ($i++$), and loop body.
2) *Use Integer Types for Loop Counters*: Loop counters should be of an integer type rather than a floating-point type.
3) *Ensure Finite Iterations*: All loops should be properly terminated, with loop-controlling expressions that are neither constants nor invariants. Include an iteration expression, and ensure that looping iterations are finite.
4) *Boolean Evaluation in Loops*: Controlling expressions in loops must evaluate to a boolean expression.
5) *Proper Loop Bodies*: The bodies of loops must be enclosed within a block.
6) *Isolate Side Effects*: A full expression containing an increment (++) or decrement (−−) operator should have no other potential side effects.
7) *Proper Conditional Bodies*: The bodies of conditionals must be enclosed within a block.
8) *Avoid Variable Wrapping in Loops*: Avoid wrapping iteration variables in loop counters.
9) *Mandatory Else Statement*: Include an `else` statement in `if` conditional structures, even when using `else-if` branches.
10) *Form Well-Structured Switch Statements*: `switch` statements, used as an effective alternative to long `if-else-if` chains for pattern matching, should be well-formed. Avoid cases that fall outside the expected value or label's scope.
11) *Include Breaks in Switch Statements to Prevent Fall-Through*: In languages that allow fall-through in `switch` statements, always include `break` expressions.
12) *Include a Default Switch Statement Case*: Include a default case in `switch` statements to handle unmatched cases. Place

the default case in last of the switch block similar to `else` condition.

13) *Use Multiple Patterns for Switch Statements*: Only use `switch` statements when there are at least two patterns to match, excluding the default case.
14) *Avoid Boolean Match Cases for Switch Statements*: Do not use `switch` statements with boolean values as cases. Instead, use simple `if-else` statements.
15) *Ternary for Simple Conditions*: Use the ternary conditional operator for simple conditions, and always use `if-else` conditions for complex ones.
16) *Ternary Operator Rule*: When using the ternary conditional operator, the first operand should evaluate to a boolean.
17) *Avoid Assignment in Conditionals*: Do not include assignment expressions within conditional evaluation expressions.
18) *Type Consistency in Conditionals*: Ensure type consistency in the operands of conditional expressions and use casting if necessary.
19) *Comparison Operators in Loop Counter*: If a loop counter is not modified using increment (++) or decrement (--) operators, it shall only be used as an operand in comparison operations within the loop condition.
20) *Ensure Finite Looping*: Ensure that the loop counter terminates and does not result in infinite looping.
21) *Use Loops only if*: Loops should only be utilized if iteration is required more than twice. Avoid higher-level constructs and control flow constructs for simplified direct execution.
22) *Switch Case Logic*: Never include logical expressions or declarations inside the switch case statement before the first case label.
23) *Conditional Control Flow*: Use switch statements or equivalent conditional control flow for executing simple instructions; for complex logic, utilize functions over switch statements.
24) *Modify collections during iteration*: In a for-each or a for-all loop, do not modify a collection's element by writing to or removing it during loop iteration.

H. External Inputs

1) *Validate External Inputs*: All external inputs received from outside the application must be rigorously validated, including checks on range, data structure, length, and types.
2) *Isolate Validation Logic*: External input validation must be isolated and separated from loosely coupled codebases to ensure that it does not impact the main application logic.
3) *Handle Invalid Inputs Safely*: If an invalid external input is detected, it must be safely handled using `Result` type.
4) *Invalid Inputs Audit*: Document or log all invalid sanitized inputs for debugging and auditing purposes.
5) *Pass Valid Inputs Securely*: Valid external inputs can be passed through interface functions of loosely coupled codebases.
6) *Prevent Injection Attacks*: Valid external inputs should not contain special or unknown characters that could be exploited via injection attacks or other security vulnerabilities.
7) *Avoid Shell Injection Risks*: Avoid executing direct shell commands from external input arguments. Instead, use safer command methods that treat inputs as elements of commands to mitigate shell injection attacks.
8) *Canonicalization and Sanitization*: URLs as external inputs should be canonicalized and properly sanitized.
9) *Ensure Input Bounds*: Always ensure that input data is bounded.
10) *Validate non-user generated inputs*: Validate non-user generated inputs such as hidden fields.
11) *Revalidate parser data*: Data received from parsers should be revalidated to only allow valid input.
12) *Inspect permissions*: Inspect and revoke any excessive permissions or privileges from external inputs before de-serialization.

I. Error Handling

1) *Prefer Error Handling Over Termination*: Always prefer error handling mechanisms over termination functions.
2) *Minimum Exception Expectation*: There should be at least one exception handling expectation per function.
3) *Error Handlers*: Exception or error handlers on functions should be structured from most specific to least specific (most general).
4) *Concise Error Handling Models*: Keep error handling simple and concise; avoid catch-all statements and define all instances of error possibilities.
5) *Exception Handling*: Exceptions or errors should be handled within their scope.
6) *Nested Exceptions*: Operations should not cause nested or additional exceptions or errors.
7) *Type Conversions*: All type conversions should include error handling mechanisms.
8) *Avoid logging in error handling*: Avoid using error or exception handling mechanisms while logging information.
9) *Integer range for byte sequences*: Integers used for writing byte sequences in any operations should be written within the range of 0 to 255.
10) *Resource Release*: Resources should be properly released when catching errors/exceptions to avoid leakage.
11) *Classifiable Error Status*: Ensure error status mixed with resulting values of operations or functions should be clearly classifiable.
12) *Proper Error Handling*: Ensure system errors are properly handled.

J. Memory

1) *Prefer Ownership or Garbage Collection Models*: Use memory management models with ownership (e.g., Rust, Mojo) or automatic garbage collection (e.g., Go, Java) favorable for dynamic-sized types stored on the heap.
2) *Avoid Manual Memory Control*: Do not perform manual memory management tasks such as initialization of variables, deletion, memory allocation, or freeing. Rely on automatic management mechanisms provided by the language compiler.
3) *Ensure Proper Resource Closure*: All resources utilized by the high-level language—such as memory (stack or heap), files, network connections, threads, locks, hardware resources, timers, and GUI resources—must be properly closed or freed in the correct order (from open to close) or a well-defined automatic closing mechanism should be available to free resources.
4) *Conditional Closure*: Resource's state must be closed or freed only if they have been successfully opened or initialized, ensuring that no attempt is made to close, free or utlize a resource in any operations which was never opened.
5) *Resource Scope Design*: Every resource's scope should be clearly designed for initialization and closure.
6) *Dangling Reference of a Freed Resource*: Avoid dangling references by not returning a reference to an automatic/local variable that is freed after the function terminates.
7) *Ensure Non-Overlapping Memory*: When cloning and copying objects, always ensure that the memory is not being overlapped. By default, avoid memory manipulation techniques to eliminate the potential for such errors.

8) *Separate Memory Locations*: Different data types should occupy distinct memory locations and must not overlap.
9) *Protect Memory Locations*: Arguments with memory locations should not be overwritten before they are utilized or passed to a function.
10) *Prefer Safe Abstractions*: Prefer using safe abstractions over direct memory/pointer manipulation functions.
11) *Avoid Manual Destructors*: Do not manually clean up resources using destructors or functions; rely solely on automatic memory management mechanisms.
12) *Resource Scope Management*: Use the minimum scope of resources possible; do not extend the lifetime of active resources if not actively used.
13) *Atomic File Operations*: Atomic operations should be utilizied on files being modified; avoid I/O functions on modified file streams.
14) *Memory-Safe Cleanup*: Utilize memory-safe cleanup functions provided by memory-safe languages.
15) *Avoid Dangling References as Return*: When a function returns a reference to a data type, ensure that the original resource or memory is not freed or deallocated after the function terminates, to prevent dangling references and undefined behavior.
16) *String Memory Allocation*: A language's string memory allocation should have space for character data and null terminator.

K. Concurrency

1) *Use Thread-Local Storage*: When multiple instances of a function run simultaneously, use thread-local storage to ensure that each thread has its own instance of a variable. Ensure each function instance has its own memory.
2) *Design Functions Without Shared State*: Design functions to avoid relying on shared state or variables, thus avoiding data races by default.
3) *Thread-Locks for Shared Memory*: When shared memory is necessary and known at compile time, use thread locks to manage concurrent access and prevent data races.
4) *Avoid Concurrent Operations on Void or Null Data*: Concurrent operations must not be performed on void data or null values.
5) *Avoid Direct Member Access*: During concurrent access to structures, avoid directly accessing individual members. Instead, perform operations on the entire object.
6) *Order of Operations and Thread Safety*: The return value of expressions and their side effects shall be independent of the order of operations and thread interleaving.
7) *Evaluation Order*: Ensure that expressions yield consistent results regardless of evaluation order or thread execution sequence.
8) *Avoid Concurrent Access Conflicts*: Do not access the same resources for read and write operations across different instances.
9) *Use Synchronization Mechanisms*: Employ synchronization mechanisms such as locks, atomic operations, or similar functions to ensure that memory operations are performed in a sequentially consistent order.
10) *Prevent Resource Leaks*: Avoid resource leaks by properly releasing or deallocating shared resources that are no longer needed or are out of scope. To automate this process, prefer automatic memory management over manual memory management.
11) *Avoid Data Races with Immutable Values*: Prefer reading from immutable values to avoid data races.
12) *Variable Scope*: Local or Thread-local variables should not be accessed outside their defined scope to prevent undefined memory behavior.
13) *Atomic Lock Abstractions*: Use higher-level abstractions for atomic locks that manage locking internally for concurrent data access; avoid simple locks that may lead to deadlocks.
14) *Thread-Safe Mechanisms*: Always prefer thread-safe mechanisms for managing thread lifecycles, avoiding signals for terminating threads.
15) *Avoid Asynchronous Cancellation*: Avoid using threads that can be canceled asynchronously due to risks of inconsistent states, resource leaks, and deadlocks. Instead, adopt cooperative cancellation strategies.
16) *Lock and Access Rules*: When a thread holds a lock for a specific data element or part of a data structure, ensure that no other threads access adjacent data concurrently.
17) *Automated Storage Duration*: Use automated storage duration models for thread locks offered by safe languages that provide automated memory management.
18) *Predefined Lock Order*: Lock atomic variables in a predefined order to minimize the risk of deadlocks.
19) *Single Lock Per Operation*: Do not use more than one lock per conditional operation by a thread.
20) *Module Lock Acquisition*: Locks used for concurrency or accessing shared resources should be acquired and released in the same module and at the same level of abstraction.
21) *Safe Concurrency Methods*: Prefer safe concurrency methods over unsafe methods to access shared resources.
22) *Synchronization Mechanisms*: Use synchronization mechanisms to ensure that changes made by one thread to a shared resource are visible to other threads accessing it.
23) *Synchronization Order*: Always properly acquire locks, modify values, and release locks in the correct order.
24) *Thread Cleanup*: Always join and detach threads regardless of their exit status to ensure proper cleanup of resources.
25) *Avoid Indefinite Waiting*: Avoid performing conditional operations on shared resources that may cause other threads to wait indefinitely for the lock to be released.
26) *Mutex Lifetime Management*: A mutex which provides exclusive access to shared resources should outlive the data it protects.
27) *Atomic Operations Requirement*: All operations, whether compound or simple, on shared resources must be atomic.
28) *Non-Lock Strategies*: When attempting concurrent resource access without locks, include strategies such as versioning to avoid concurrency issues.
29) *Mutex Destruction Safeguards*: A mutex should be fully unlocked before destroying it to avoid undefined behavior.
30) *Synchronization for Bit-Fields*: Use proper synchronization mechanisms, such as locks or atomic operations, to prevent data races when accessing bit-fields from multiple threads.
31) *Deadlock Prevention*: Avoid deadlocks by locking in a predefined order.
32) *Condition Validation After Wakeup*: Conditional threads should check the condition after waking up to ensure it is still valid, mitigating spurious wake-ups.
33) *Prevent Starvation*: Avoid unnecessary waiting conditions to prevent scenarios where threads become starved or blocked indefinitely.
34) *Atomic Variable Reference*: Always avoid referring to an atomic variable twice in a single expression.
35) *Lock Release*: Locks should be released in exceptional or error conditions.
36) *Mutex Relocking*: Avoid relocking a mutex that is owned by the current thread.
37) *Locking and global mutable variables*: Locking and concurrent

logic should be implemented in the file's scope of global mutable variables that are utilized in other codebases.
38) *Resource availability before locking*: Ensure resource availability before acquiring a lock and before initializing it.
39) *Avoid client-defined locking patterns*: Avoid client-side inputs that define locking patterns. Always manage concurrency internally in the codebase.
40) *Individual thread management*: Safely manage threads individually instead of using group management methods.
41) *Notify threads based on priority*: Notify waiting threads individually based on priority for shared resources.
42) *Locking strategies for shared access*: Shared access to collection data structures or types should prefer locking the entire structure for frequent read operations, and granular element locks for frequent write operations.
43) *Prevent simultaneous operations*: Prevent simultaneous read and write operations on shared resources.
44) *Awaiting threads in loops*: Involve loops for awaiting threads that access shared resources.
45) *Graceful thread termination*: Do not abruptly terminate threads; instead, use flags or interruption mechanisms to notify the thread to stop.
46) *Prefer bounded thread pools*: Prefer bounded thread pools, i.e., predefined threads over unbounded ones.
47) *Interdependent tasks in thread pools*: Interdependent tasks in a bounded thread pool should have enough threads to complete intended operations.
48) *Incorruptible concurrent operations*: Concurrent operations should allow only incorruptible executions, enabling proper closure of resources.
49) *Error handling in thread operations*: Thread operations should include error handling mechanisms.
50) *Thread-local variable reinitialization*: Thread-local variables should be reinitialized for each thread in the thread pool.
51) *Prevent access before initialization completion*: Do not allow background threads to access initialized constructs before their initialization is complete.

L. I/O

1) *File Operations on Device Files*: Avoid performing file operations (like reading, writing, or seeking) on hardware device files.
2) *Valid Character Distinction*: While performing operations on files, distinguish between valid characters from the file and End of File markers.
3) *Check Return Values*: When reading data from files, always check that the return value indicates success, and do not assume the read data is non-empty. Handle empty data or escape sequences appropriately.
4) *Reset Buffers on Failure*: Reset buffers if a file read operation fails to ensure clearing potentially unsafe leftover data.
5) *File Identification Using Hashes*: Instead of relying solely on file names for identification, consider using a hash of the file's content or metadata for a more stable means of identification.
6) *Canonicalize Path Names*: Canonicalize path names of files.
7) *Access Permissions for Files*: Create files with appropriate access permissions to safeguard sensitive information.
8) *File Closure Before Removal*: A file should be closed before removing it from the file system, additionally ensure all its supplementary files/attributes related to it are also being removed.
9) *File Name Conflict Checks*: Check for existing file name conflicts and ensure the file is closed before executing a renaming operation.
10) *Single Character Re-Processing*: Re-process a single processed character (pushback) by the stream/buffer instead of multiple re-processing of characters in a file operation.
11) *Format-Specific Operations*: File operations should be specific to their formats (e.g., text file format may have special sequences of standards, binary file format includes raw sequential bytes).
12) *Secure File Operations Directory*: Conduct file operations only within a secure and trusted directory.
13) *Dedicated Temporary File Directory*: Use a dedicated temporary file directory; never use a shared location.
14) *Link Existence Check*: If file operations involve links to directories, ensure that the link exists before opening or executing operations on the file object.
15) *Sandboxing Techniques*: Use techniques such as jails, containers, or sandboxing to confine applications to specific directories and reduce system-wide vulnerability induced by the application.
16) *Symbolic Link Operations*: While operating on a file given by a symbolic link, ensure proper checks and validations are in place.
17) *Revocation of Privileges*: While revoking privileges, ensure there are no pending operations that utilize those privileges, start with the least critical privileges, and ensure privileges are successfully revoked.
18) *Data Transfer Integrity*: To ensure correct data transfer between systems, always consider it's byte order, i.e., little or big-endian serialization.
19) *Specialized Functions for I/O*: Use compatible specialized functions for I/O/FILE operations rather than utilizing commonly available functions.
20) *Handle file-related errors*: Handle file-related errors thrown by the underlying file system or environment.
21) *Remove temporary files on termination*: Ensure temporary files are properly removed during safe termination.
22) *Environment Variable Validation*: Validate and handle environment variables appropriately to avoid size-related issues. Implement an arbitrary safe limit on size assumptions of environment variables.
23) *Mitigating Environmental Conflicts*: Mitigate conflicting environmental variables with the same name.
24) *File Object References*: Return a reference to file objects instead of copying them to avoid resource management issues.

M. Testing

1) *Mandatory Test Cases*: Every codebase must include comprehensive test cases.
2) *Unit Tests for Functions*: Implement unit tests for every function within loosely coupled codebases.
3) *Integration Tests for External Inputs*: Use integration tests to validate the handling of external inputs across multiple interface functions.
4) *Test High Cohesion Functions*: All high-cohesion functions within loosely coupled codebases should include test cases for all known outputs, including all possible variants of enumerations. This requirement applies to wrapper functions as well.
5) *Result and Option Return Types*: Functions that return `Result` types (e.g., `ok` or `err`) or `Option` types (e.g., `none` or `some`) must be tested for all possible variants.
6) *Assertion Validation*: Never rely on assertions compared to function-specific test cases.

7) *Assertions without side effects*: Assertions that check the correctness of the code should not produce side effects that modify the state of the code.

N. Languages

1) *Primary Language Requirement*: The repository must specify a primary programming language, which should be publicly documented.
2) *Compile-Strict Languages* Prefer languages that offer compile-time evaluation keywords and have strict compilers.
3) *Polyglot Rule*: Code written in secondary high-level languages must be placed in separate files and imported into the primary language's codebase.
4) *Encapsulation of Polyglot Methods*: Polyglot methods must be encapsulated in isolated modules or codebases.
5) *Separation of Hardware-Specific Code*: Hardware-specific code, such as assembly language or binary instructions, must be placed in separate files and imported into the primary language, similar to the polyglot rule.
6) *Variant Management*: Hardware-specific code must have variants for each target environment.
7) *Achieve Zero Compiler Warnings*: Ensure that the codebase compiles with zero warnings to meet the highest safety standards.
8) *Adhere to Language Standards*: Follow the rules defined in the official language documentation and adhere strictly to language standards.
9) *Intended Use of Identifiers*: Use predefined or language-reserved identifiers for their intended use according to their intended syntax.

O. Libraries

1) *Avoid Duplicate Inclusions*: Prevent iterative compilation issues by ensuring that the inclusion of source code or libraries is not duplicated.
2) *Evaluate Imported Libraries*: Carefully assess imported libraries based on their usage history in mission-critical applications, their developers, and other relevant factors to ensure safety and compliance.
3) *Use Wrapper Functions*: Use newly defined wrapper functions instead of calling library functions directly.
4) *Validate External Library Data*: Validate data returned by external libraries before use.
5) *Use Package Managers*: Prefer languages with package managers that handle importing libraries and external modules automatically, to prevent manual import inconsistencies.
6) *Check Conflicts Identifiers*: Avoid importing codebases that may include conflicting identifiers; ensure the imported codebase is conflict-free.
7) *Memory and Thread Safety*: External libraries should be memory and thread-safe and follows the ESS guidelines.
8) *Mathematical Libraries*: Use mathematical libraries that handle domain and range errors internally; validate inputs before passing to mathematical functions and implement appropriate error handling.
9) *Error-Handling Mechanisms*: Use only libraries that implement error-handling mechanisms; avoid any equivalents that do not.
10) *Compatible Error-Handling Strategies*: Use libraries that provide error detection without dictating specific error handling strategies that might be incompatible.
11) *Standard Library Error Handling*: Handle standard library errors efficiently.

P. Trust

1) *Prevent untrusted execution termination*: Never allow an untrusted execution to terminate the program.
2) *Serialization/sanitization procedures*: Serialization/sanitization procedure applied to any resource or data structure before passing to untrusted execution should prevent exposure and mitigates risks for the trusted execution or its state.
3) *Trusting Untrusted Code*: Never assume untrusted code may not misuse its offered privileges.
4) *Trust hierarchy in execution*: Untrusted code execution should only be followed by trusted code execution.
5) *Pass copies of data*: Pass copies of data instead of references when sharing data with untrusted executions.
6) *No hard-coded sensitive information*: Do not hard-code sensitive information. Design protocols that prefer avoiding constants or literals.
7) *Sensitive data uniqueness and immutability*: Sensitive data should remain unique and immutable.
8) *Access immutable sensitive data*: Access immutable sensitive data via references to encapsulate and prevent cloning or unintended operations.
9) *Senstive Data with Bounds*: avoid sharing sensitive data without explicitly defined bounds.
10) *Sensitive Data Positioning*: Operations or functions that operate on sensitive data must be positioned at the beginning of the memory layout and should not be followed by non-sensitive data.
11) *Sandboxing Sensitive Operations*: Keep sensitive operations isolated in a sandboxed environment with minimal access to required operations only.
12) *Sensitive Information Handling*: Do not log or cache sensitive information.
13) *Minimize Sensitive Data Lifetime*: Minimize in-memory lifetime of raw sensitive data.
14) *Sensitive Information Protection*: Protect sensitive information (such as passwords, encryption keys, or personal data) by ensuring it is not inadvertently written to disk.
15) *Memory allocation for sensitive information*: Sensitive information should remain within its allocated memory space.
16) *Do not serialize unencrypted sensitive data*: Do not serialize unencrypted sensitive data.
17) *Sensitive-Encrypted Data Storage*: Store only encrypted sensitive data on disk or persistent storage.
18) *Use provable secure cryptography*: Use only provable secure cryptography for data exchange and tamper-proof authentication.
19) *Cryptographic signatures on sensitive operations*: Use cryptographic signatures only on code that performs privileged and sensitive operations.
20) *Validate codebases for tamper-proofing*: Use cryptographic signatures and hashes for tamper-proofing codebases.
21) *Cryptographic Data Exchange*: Use cryptographic primitives for data exchange of sensitive information between untrusted modules or networks.
22) *Use cryptographic signatures*: Use cryptographic signatures on serialized data when passing it beyond a trusted boundary.
23) *Weak Cryptographic Algorithms*: Avoid potentially weak cryptographic algorithms/functions that are proved to be breakable under polynomial time.
24) *Use Cryptographically Secure RNGs*: Random number generator function should be cryptographically secure and provides provable randomness.

Q. Others

1) *No Unsafe Code*: Do not use unsafe constructs (e.g., unions) under any circumstances.
2) *Number Systems*: Use only decimal and hexadecimal number systems. Avoid using octal.
3) *Avoid Type-Unsafe Constructs*: Avoid any type-unsafe constructs, this includes variadic functions, incompatible type casting, pointers, textual substitutions, etc.
4) *Prefer Procedural over Polymorphism*: Do not use polymorphism, class, or inheritance-based approaches to achieve code reusability. Instead, use a procedural programming i.e., functional approach.
5) *Compile-Time Checks*: Prefer compile-time checks to runtime checks, as they evaluate the program before execution.
6) *Respect Program Constraints*: Do not violate any program-specific constraints defined in the codebase.
7) *Safe termination logic*: Define safe termination logic with proper closure of existing resources within the program to handle both internal and external termination requests.
8) *Non-local Jumps*: Avoid non-local jumps that disrupt control flow; use linear-structured control flow instead.
9) *Avoid overriding safe constructs*: Do not override safe constructs or operations with unsafe constructs or operations.
10) *Avoid runtime structure inspection*: Avoid inspecting or altering a program's structure at runtime.
11) *Order of Evaluation*: Avoid relying on order of evaluation for evaluating side effects.
12) *Efficient Tokenization*: Tokenization of strings should be efficiently handled with delimiters, and that the original string is not modified.
13) *Bounds-Checking for Strings*: Use bounds-checking interfaces for string manipulation to prevent buffer overflows.
14) *Valid Indices*: Do not use or access any value or indices that are outside their defined ranges.
15) *Serialize data from immutable objects only*: Serialize data from immutable objects only.