ISO IEC JTC 1 SC 22 WG 23 N0629
19 January 2016

Rules to avoid programming language vulnerabilities that apply to most common languages

Meta-rule: Develop and use a coding standard based on this document that is tailored to your risk environment.

1. Validate input. Do not make assumptions about the values of parameters. Check parameters for valid ranges and values in the calling and/or called functions before performing any operations.
2. Check return values from subprograms.
3. Enable compiler static analysis checking and resolve compiler warnings.
4. Run a static analysis tool.
5. Perform range checking.
6. Allocate and free memory at the same level of abstraction.
7. Test constructs that have unspecified behavior for all possible behaviours.
8. Ensure that undefined or deprecated language features are not used.
9. Error detection, reporting, correction, and recovery should be an integral part of a system design.
10. Use only those features of the programming language that enforce a logical structure on the program.
11. Sanitize, erase or encrypt data that will be visible to others (for example, freed memory, transmitted data).
12. Develop and use a coding standard based on this document that is tailored to your risk environment.

Here are possible additions as these are mentioned in specific guidance across multiple languages or apply to most common languages:

- Avoid using features of the language which are not specified to an exact behaviour. The abundant nature of implementation-defined behaviour makes it difficult to avoid. As much as possible users should avoid implementation defined behaviour. Document instances of use of unspecified behaviour. Code that makes assumptions about the unspecified behaviour should be replaced to make it less reliant on a particular installation and more portable.
- Avoid using libraries without proper signatures
- Do not modify loop control variables inside the loop body
- Do not perform assignments within Boolean expressions.
- Do not depend on side-effects of a term in the expression itself
- Use names that are clear and visually unambiguous. Be consistent in choosing names.
- Use careful programming practice when programming border cases.
- Be aware of short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression such as if the first expression evaluates to `false` in an and expression, then the remaining expressions, including functions calls, will not be evaluated.

- It is best to avoid fall-through from one case statement into the following case statement but if necessary then provide a comment to inform the reader that the fall-through is intentional.
- Do not use floating-point arithmetic when integers or booleans would suffice.

Rules to avoid programming language vulnerabilities in Ada

1. Do not use features explicitly identified as unsafe, such as Unchecked_Deallocation or Unchecked_Conversion.
2. Handle all Exceptions raised by type and subtype-conversions.
3. Protect all data shared between tasks within a protected object or mark the data Atomic.
4. Use pragma Atomic and **pragma** Atomic_Components to ensure that all updates to objects and components happen atomically.
5. Use pragma Volatile and **pragma** Volatile_Components to notify the compiler that objects and components must be read immediately before use as other devices or systems may be updating them between accesses of the program.
6. Rather than using predefined types, such as Float and Long_Float, whose precision may vary according to the target system, declare floating-point types that specify the required precision (for example, digits 10). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.
7. Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as 'Exponent).
8. For **case** statements and aggregates, do not use the **others** choice.
9. Use Ada's support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing.

Rules to avoid programming language vulnerabilities in C

1. Make casts explicit in the return value of malloc. (ref to HFC??)

   Example: s = (struct foo*) malloc(sizeof(struct foo));

   uses the C type system to enforce that the pointer to the allocated space will be of a type that is appropriate for the size. Because malloc returns a void *, without the cast, "s" could be of any random pointer type; with the cast, that mistake will be caught.
2. Use length restrictive functions such as strncpy(), strncmp(), and strncat(), snprintf(), instead of strcpy(), strcmp and strcat, sprintf(), respectively. When substituting strncpy for strcpy, ensure that the result will always be null-terminated. Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces.
3. Use commonly available functions such as htonl(), htons(), ntohl() and ntohs() to convert from host byte order to network byte order and vice versa. [6.3]
4. Use stack guarding add-ons to detect overflows of stack buffers.
5. Perform range checking before accessing an array or before calling a memory copying function such as memcpy() and memmove() since bounds checking is not performed automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
6. Create a specific check that a pointer is not null before dereferencing it. As this can be expensive in some cases (such as in a for loop that performs operations on each element of a

large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.

7. Set a freed pointer to null immediately after a free() call, as illustrated in the following code:
    i. free (ptr);
    ii. ptr = NULL;
8. Do not use memory allocated by functions such as malloc() before the memory is initialized as the memory contents are indeterminate.
9. Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible. Any of the following operators have the potential to wrap or have undefined behavior in C:
    ```
    a + b     a – b     a * b    a++      a--
    a += b    a -= b    a *= b   a << b   a >> b  -a
    ```
10. Do not modify a loop control variable within a loop. Even though the capability exists in C, it is still considered to be a dangerous programming practice.
11. Check the value of a larger type before converting to a smaller type to see if the value in the larger type is within the range of the smaller type.

Rules to avoid programming language vulnerabilities in Python

1. Do not use floating-point arithmetic when integers or booleans would suffice.
2. Use of enumeration requires careful attention to readability, performance, and safety. There are many complex, but useful ways to simulate enums in Python [(Enums for Python (Python recipe))]and many simple ways including the use of sets:
    ```
    colors = {'red', 'green', 'blue'}
    if  red  in colors: print('valid color')
    ```
    Be aware that the technique shown above, as with almost all other ways to simulate enums, is not safe since the variable can be bound to another object at any time.
3. Ensure that when examining code that you take into account that a variable can be bound (or rebound) to another object (of same or different type) at any time.
4. Avoid implicit references to global values from within functions to make code clearer. In order to update globals within a function or class, place the global statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are global (for example, global a, b, c).
5. Use only spaces or tabs, not both, to indent to demark control flow. Never use form feed characters for indentation.
6. Use Python's built-in documentation (such as docstrings) to obtain information about a class' method before inheriting from it.
7. If coding an extension utilize Python's extension API to ensure a correct signature match.
8. Either avoid logic that depends on byte order or use the sys.byteorder variable and write the logic to account for byte order dependent on its value ('little' or 'big').
9. When launching parallel tasks don't raise a BaseException subclass in a callable in the Future class.
10. Do not depend on the way Python may or may not optimize object references for small integer and string objects because it may vary for environments or even for releases in the same environment.

Rules to avoid programming language vulnerabilities in Ruby

1. Use the language's built-in mechanisms (rescue, retry) for dealing with errors.
2. Use symbols for enumerators rather than named constants.
3. Provide code to catch exceptions resulting from mismatches between objects and methods.
4. Knowledge of the types or objects used is a must. Compatible types are ones which can be intermingled and convert automatically when necessary. Incompatible types must be converted to a compatible type before use.
5. In most cases a break statement can be avoided by using another looping construct. These are abundant in Ruby.

Rules to avoid programming language vulnerabilities in Spark

1. Rather than using predefined types, such as Float and Long_Float, whose precision may vary according to the target system, declare floating-point types that specify the required precision (for example, digits 10). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.
2. Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as 'Exponent).
3. For **case** statements and aggregates, do not use the **others** choice.

Rules to avoid programming language vulnerabilities in PHP

1. Be cognizant that arithmetic for integers that exceed their bounds becomes floating point math which may not have the exact same behaviour.
2. Test the implementation in use to see if exceptions are raised for floating point operations and, if they are, then use exception handling to catch and handle wrap-around errors.
3. Be careful when retrying an operation after an exception to avoid an endless loop.
4. Use PHP's error handling functions and/or `Exception` class to implement an appropriate termination strategy.
5. Utilize the provisions in the Zend framework to configure extensions so that all parameters are accurately and completely specified.
6. If coding an extension utilize PHP's extension API to ensure a correct signature match.
7. Utilize PHP's rich library of string filtering "sanitize" functions to screen the program's logic from malformed input strings.
8. Do not depend on the way PHP may or may not compare strings that contain long integers.
9. Set `error_reporting` to enable the `E_DEPRECATED` and `E_USER_DEPRECATED` bit masks to warn about the use of any deprecated language constructs or functions.
10. Ensure that when examining code to take into account that a variable can be bound (or rebound) to another object (of same or different type) at any time.

Rules to avoid programming language vulnerabilities in Fortran

1. Never use implicit typing. Always declare all variables. Use `implicit none` to enforce this.
2. Use explicit conversion intrinsics for conversions of values of intrinsic types, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.

3. Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used. Similarly, use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
4. Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
5. Obtain array bounds from array inquiry intrinsics wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
6. Use default initialization in the declarations of pointer components.
7. Specify `pure` (or `elemental`) for procedures where possible for greater clarity of the programmer's intentions.
8. Code a status variable for all statements that support one, and examine its value prior to continuing execution for faults that cause termination, provide a message to users of the program, perhaps with the help of the error message generated by the statement whose execution generated the error.
9. Avoid the use of common and equivalence. Use modules instead of common to share data. Use allocatable data instead of equivalence.
10. Supply an explicit interface to specify the `external` attribute for all external procedures invoked.