ISO IEC JTC 1 SC 22 WG 23 N0626
19 January 2016

Proposed top PHP specific guidance:

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.

For reference, here is all of the PHP specific guidance from 24773 (n0461):

- Avoid rebinding variables to a different type except where it adds value.
- 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.
- Be aware that when PHP performs bitwise operations on strings it does so using the ASCII value of each character.
- Apply the general guidance from Section 6.5.
- Be aware that results will frequently vary slightly by implementation (See ***Error! Reference source not found.*** for more on this subject).
- If higher precision is required use PHP's `gmp` functions or arbitrary precision math functions.
- Use the `default` clause (possibly with error handling/reporting logic) when all cases are not covered.
- Comment "fall throughs" to make it clear to the reader that it's intentional.
- Use explicit casts when it makes the code clearer.
- Pay special attention to issues of magnitude and precision when using mixed type expressions.
- Although string access violations will not cause buffer overflows, they can cause unexpected behaviors so consider using bounds checking whenever using indexes from untrusted sources.
- Be cognizant that arithmetic for integers that exceed their bounds becomes floating point math which may not have the exact same behaviour.

- Ensure that integers used in loop control have not been converted to floats. See ***Error! Reference source not found.***.
- 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.
- Do not assume that an over or underflow halt will occur when shifting using bitwise operators; use higher level functions such as multiplication and division when that is the intent or use functions from the `gmp` extension.
- Do not assume the bit orientation of the hardware stores bits left-to-right or right or right to left. If the program logic depends on the direction bits are stored then be aware that results will differ based on the platform used.
- Avoid names that differ only by case unless necessary to the logic of the usage.
- Do not use overly long names.
- Use names that are not similar (especially in the use of upper and lower case) to other names.
- Use meaningful names.
- Use names that are clear and visually unambiguous.
- Remove assignments to all variables that are never used.
- Do not use identical names unless necessary to reference the correct object.
- Avoid the use of the global and nonlocal specifications because they are generally a bad programming practice for reasons beyond the scope of this document and because their bypassing of standard scoping rules makes the code harder to understand.
- Use namespaces to differentiate functions, constants, classes, and interfaces from global names and names within included files.
- Make certain the rules for namespaces are well understood to avoid inadvertently referencing the wrong item.
- Ensure that it is not logically possible to reach a reference to a variable before it is assigned. The example above illustrates just such a case where the programmer wants to print the value of `$a` but has not assigned a value to `$a` – this proves that there is missing, or bypassed, code needed to provide `$a` with a meaningful value at runtime.
- Use parenthesis liberally to force intended precedence and increase readability.
- PHP does not guarantee the order of evaluation for sub expressions so break large/complex statements into smaller ones using temporary variables for interim results.
- Be aware of PHP's short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression; if necessary perform each expression first and then evaluate the results:

  ```
  <?php
  $x = a();
  $y = b();
  if ($x or $y) …
  ?>
  ```
- Note that when evaluating and expressions (`&&`), if the first expression evaluates to `false` then the remaining expressions, including functions calls, will not be evaluated.

- Move assignments outside of Boolean expressions.
- Do not confuse the equivalence operator (==) with the assignment operator (=).
- Bound not (`!`) operations with parenthesis.
- Simplify overly complex expressions.
- Do not use assignment expressions is function parameter lists.
- 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.
- Generally speaking the `default` case should be used for handling all unexpected cases which should then be treated as errors.
- Use `end-if` (or similar) statement to demark the end of constructs.
- Use indentation to clarify and use pretty print programs and/or static analysis tools to check that the demarcation is as intended.
- Consider using the curly brace to demark blocks.
- Do not modify a loop control variable within a loop. Even though the capability exists in PHP, it is still considered to be a poor programming practice.
- Be aware of PHP's indexing from zero and code accordingly.
- Avoid using the `goto` statement other than to exit multi-level loops and even then branch to a label near the end of the loop.
- Judicious use of `break` statements is encouraged to avoid confusion.
- When practical, and the objects being passed are small, use call by copy to minimize the chance that the called function can cause damage to the passed objects.
- Though PHP supports variable argument lists it almost always clearer to have the function specify explicit parameters and have the caller use those.
- Consider using an array when a function is designed to operate on a variable number of items.
- Minimize the use of recursion and limit it to no more than 200 levels if practicable.
- When considering the use of recursive functions consider the effect that the stack and allocation/de allocation of local variable will have on performance and, if significant, consider coding a non-recursive solution.
- Use PHP's exception handling with care in order to not catch errors that are intended for other exception handlers.
- Be sure to test for the lowest level child class of `Exception` (as in the example above) when using the `catch` statement otherwise, since all exception classes are subclasses the `Exception` class.
- Handle exceptions as close to the origin of the exception when practicable to make it easier for the reader to see how an exception will be handled.
- Be careful when retrying an operation after an exception to avoid an endless loop.
- Avoid reducing the system's default level of error reporting especially during development since PHP is able to report on many questionable coding practices that can help point out potential future problems.

- Consider setting `error_reporting` to `E_STRICT` while in development mode to help catch coding errors which are not necessarily fatal but could be indicative of poor or dangerous coding practices that could cause problem in the future.
- Do not disable error checking.
- Code with a three level approach:
    - Use coding practices which help prevent, detect, and handle faults.
- Use PHP's error handling functions and/or `Exception` class to implement an appropriate termination strategy.
- Use exception handling, but directed to specific tolerable exceptions, to ensure that crucial processes can continue to run even after certain exceptions are raised.
- If a function can fail consider using a return code to indicate the caller the kind of error that has occurred.
- If an exception renders further execution impossible then wrap up all processing in a manner that releases resources (close files and so on) and destructs classes.
- Use PHP's numerous error detecting, reporting, and handling functions to intercept and handle errors.
- Release all objects when they are no longer required.
- Document classes.
- Inherit only from trusted classes.
- Do not make assumptions about the values of parameters.
- When practicable check parameters for valid ranges and values in the calling and/or called functions before performing any operations.
- Utilize the provisions in the Zend framework to configure extensions so that all parameters are accurately and completely specified.
- Avoid using `eval` and *never* use it with untrusted code.
- Use only trusted modules as extensions.
- If coding an extension utilize PHP's extension API to ensure a correct signature match.
- Wrap calls to library routines and use exception handling logic to intercept and handle exceptions when practicable.
- Enable as much error checking as practicable for tested, production-ready code.
- Do not suppress any error reporting during development (enable `E_STRICT` to see the best advice including which functions are deprecated).
- Utilize PHP's rich library of string filtering "sanitize" functions to screen the program's logic from malformed input strings.
- Ensure that a function is defined before attempting to call it.
- Be aware that a function is defined dynamically so its composition and operation may vary due to variations in the flow of control within the defining program.
- Do not depend on the way PHP may or may not compare strings that contain long integers.
- Avoid the shown usages of the constructs above.
- Design and test usage of these functions to utilize them in a way that produces consistent results regardless of machine size or avoid their use completely if possible.

- 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.