ISO/IEC JTC 1/SC 22/WG 23 N 0393

Proposed Annex for PHP Language

Date	2012-03-28
Contributed by	Kevin Coyne
Original file name	PHP_Annex_v16.docx
Notes	• Initial draft (just to ensure format, sequence) • Highlighted text to be revisited for more/better explanation etc. • Red Highlighted is missing • Red text is a place holder or note to author or reviewer • Blue text denotes text copied from the Python annex as a place holder and as a guide for me as to the amount of documentation that is preferred as well as style. • PHP 5.3.8

Notes to Self

Annex PHP

PHP. Vulnerability descriptions for the language PHP Standards and terminology

PHP.1 Identification of standards and associated documents (Achour)

Bibliography

Achour, M. (n.d.). *PHP Manual*. Retrieved 3 5, 2012, from PHP: http://www.php.net/manual/en/

Brueggeman, E. (n.d.). Retrieved 3 5, 2012, from The Website of Elliott Brueggeman : http://www.ebrueggeman.com/blog/integers-and-floating-numbers

Will Dietz, P. L. (n.d.). *Understanding Integer Overflow in C/C++*. Retrieved 3 5, 2012, from http://www.cs.utah.edu/~regehr/papers/overflow12.pdf

PHP.2 General Terminology and Concepts

PHP.2.1 General Terminology

Assignment statement	Used to create (or rebind) a variable to an object. The simple syntax is `$a=$b`, the augmented syntax applies an operator at assignment time (e.g., `$a += 1`) and therefore cannot create a variable since it operates using the current value referenced by a variable. Other syntaxes support multiple targets (e.g., `$x = $y = $z = 1`).
Autoloading	
Body	The portion of a compound statement that follows the header. It may contain other compound (nested) statements.
Boolean	A truth value where `True` equivalences to any non-zero value and `False` equivalences to zero. Commonly expressed numerically as 1 (true), or 0 (false) but referenced as `True` and `False`.
Comment	There are numerous ways to comment in PHP but in this annex comments are preceded by a double hash symbol "//".
Garbage collection	The process by which the memory used by unreferenced object and their namespaces is reclaimed.
Interpolation	The substitution of the value of a variable's value when a double quoted string is evaluated.
Namespace	A place where names reside. Examples of objects that have their own namespaces include: modules, classes, and functions.
Type Juggling	Implicit casting of a value from one type to another.

PHP.2.2 Key Concepts

The key concepts discussed in this section are not entirely unique to PHP but they are implemented in PHP in ways that are not intuitive to new and experienced programmers alike.

Single versus double quotes

Alternate coding styles ({} versus :)

PHP.3 Type System [IHN]

PHP.3.1 Applicability to Language

PHP has a dynamically typed system in which variables are assigned a type at runtime. PHP is also a weakly typed language with no support for explicitly declaring types. A variable's type is determined at runtime by the value assigned to it:

```
<?php $a = 1; // $a is an integer
$a = 'x';     // $a is now a string
$a = 1.5;     // $a is now a floating point number
$b = $a;      // $b is a floating point number
?>
```

PHP provides the ability to dynamically create variables when they are first assigned a value. In fact, assignment is the only way to bring a variable into existence. The type of a variable can also change at

any time.

```
<?php
$a = 'alpha'; // assignment to a string
print "$a\n"; alpha
$a = 1.234;
print "$a\n"; 1.234
unset($a); // remove the variable
print $a; // PHP notice: undefined variable
?>
```

The PHP language, by design, allows for dynamic binding and rebinding of variables which can change a variable's type. Because PHP performs a syntactic analysis and not a semantic analysis and because of the dynamic way in which variables are brought into a program at run-time, PHP cannot warn that a variable is referenced but never assigned a value. The following code illustrates that `$c`, though never assigned a value (and thus undefined) will not generate an "undefined" notice unless executed:

```
<?php
$a = 1;
$b = 0;
if ($a > $b)
    print "\$a > \$b";
else
    print $c;
?>
```

Depending on the current value of `$a` and `$b`, an unassigned variable notice will or will not be raised for `$c`.

PHP.3.2 Guidance to Language Users

- Avoid rebinding variables to a different type except where it adds value; and
- 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; and

---

PHP.4 Bit Representations [STR]

PHP.4.1 Applicability to Language

Some interfaces require data to be passed as a series of bits in a specific length and format. The manipulation of bit strings is often required to set and/or interpret the bit strings correctly. PHP provides 6 bitwise operators but there are vulnerabilities due to machine specific characteristics such as the length, machine word boundaries, and the "endianness" of the machine.

PHP's 6 bitwise operators can each handle numbers or strings. Strings are truncated to the length of the shorter of the two operands and the operation is done on the ASCII value of each character. If given a string and a number the string is converted to a number and the operation is performed as though both

operands were numbers:

```
<?php
echo  18 & 32, "\n"; // 0
echo "18" & 32, "\n";// 0 "18" converted to integer 18 first
echo "18" & "32";    // "10" operator works on ASCII values
?>
```

As shown in the third example above, the operator, when given two strings will convert them each to their ASCII equivalent first then perform the operation on each character of each string in sequence from left to right. In this example the ordinal value of "1" (decimal 49 or 00110001), when ANDed with "3" (decimal 51 or 00110011), produces a "1" (decimal 49 or 00110001). The second character in the "18", an "8", when ANDed to the "2" in "32" produces a "0" thereby producing the string "10".

PHP.4.2 Guidance to Language Users

- Be aware that when PHP performs bitwise operations on strings it does so using the ASCII value of each character.

PHP.5 Floating-point Arithmetic [PLF]

PHP.5.1 Applicability to Language

PHP typically supports floating-point arithmetic using the IEEE 754 standard. Literals are expressed with a decimal point and or an optional `e` or `E`:

```
1., 1.0, .1, 1.e0
```

PHP.5.2 Guidance to Language Users

- Use floating-point arithmetic only when absolutely needed;
- Do not use floating-point arithmetic when integers or booleans would suffice;
- Be aware that precision is lost for some real numbers (i.e., floating-point is an approximation with limited precision for some numbers);
- Be aware that results will frequently vary slightly by implementation (See PHP.52 Implementation–defined Behaviour [FAB] for more on this subject); and
- Testing floating-point numbers for equality (especially for loops) can lead to unexpected results. Instead, if floating-point numbers are needed for loop control use >= or <= comparisons.
- If higher precision is required use PHP's `gmp` functions or arbitrary precision math functions.

PHP.6 Enumerator Issues [CCB]

PHP.6.1 Applicability to Language

The only kind of enumeration provided by PHP is the `switch` statement which is covered below. Given that enumeration is a useful programming device and that there is no enumeration construct in PHP, many programmers choose to implement their own "enum" objects or types using a wide variety of

methods including the creation of “enum” classes and functions. One simple method is to simply assign a list of names to integers. Code can then reference these “enum” values as they would in other languages which have native support for enumeration:

```
<?php
$Red = 0; $Green = 1; $Blue = 2;
$a = 1;
if ($a == $Green)
  print('$a=Green')// => a=Green;
?>
```

The disadvantage to the approach above is that any of the “enum” variables could be assigned new values at any time thereby undoing their intended role as “pseudo” constants.

The `switch` statement provides a kind of enumeration which evaluates a variable to determine which of a series of statements will be executed:

```
<?php
$x = 3;
switch($x):
  case 1:
    print('$x=1');
    break;
  case 2: // fall through to next case
  case 3:
    print('$x>1');
    break;
  default:
    print('All other values of $x');
endswitch;
?>
```

Note that “`case 2`” above falls through to “`case 3`”. That is, if the value of `$x` is `2` then the `true` path for “`case 3`” is executed. The `default` clause is used to ensure complete coverage for all possible values. Failure to specify a `default` case coupled with a presumption that all “cases” have been accounted for could lead to unexpected results.

PHP.6.2 Guidance to Language Users

- Use the `default` clause (possibly with error handling/reporting logic) when all cases are not covered; and
- Comment “fall throughs” to make it clear to the reader that it’s intentional.

PHP.7 Numeric Conversion Errors [FLC]

PHP.7.1 Applicability to Language

Conversion from one type to another is done either implicitly (known in PHP as type juggling) or

explicitly. Implicit casting for binary ***arithmetic*** operations follows a simple set of rules whenever the operands are of ***different*** types:

- When the first operand is an integer and the second is a floating point, the integer is converted to a floating point. If the second operand is a string then it's converted to a number and if that's a float then the first operand is converted to a float.
- When the first operand is a float and the second operand is a string then the second operand is converted to a float.

Note that the conversions do not convert the values of any variables on the ***right*** hand side of a statement, they create intermediate results for the purpose of evaluation as shown in the example below:

```
<?php $a = 1;
$b = 2;
$c = 1.5;
$a = $b + $c;
echo is_float($a),"\n"; // true, $a is now a float
echo is_int($b),"\n";   // true, $b is unchanged from float
echo $a;                // 3.5
?>
```

The presence of an 'e' or a period after any leading characters in a string will cause a conversion of a string to a float.

```
<?php
$a = "5" + 1;
echo $a, "\n";           // 6
$a = "5.75" + 1;
echo is_float($a), "\n"; // true
echo $a;                 // 6.75
?>
```

PHP is not type safe in that there are no provisions for causing a runtime error for invalid type usage:

- Whenever a string cannot be converted to a number (i.e., it does not start with a numeric character) the value of zero is used;
- When an array is cast to a number the value is always 1;
- When an array is cast to a string the value is the string "Array"; and
- Casting an array to an object creates an object which has one property for every key/value pair in the array.

The examples below demonstrate illogical arithmetic which causes implicit casting but do not raise any exceptions:

```
<?php
$a = "abc" + "def";
echo $a, "\n"; // 0
$a = "abc" + 34;
```

```
echo $a;        // 34
?>
```

PHP.7.2 Guidance to Language Users

- Use explicit casts when it makes the code clearer;
- Pay special attention to issues of magnitude and precision when using mixed type expressions;

PHP.8 String Termination [CJM]

PHP.8.1 Applicability to Language

There is no termination character for strings in PHP but you can address individual characters starting from an offset of zero and you append characters after the end of the string:

```
<?php
$a = 'abc';
$a{0} = 'x'; // xbc
$a{4} = 'd'; // xbc d
$a[5] = 'y'; // xbc dy  (Shows the use of [] is equivalent to {}
print($a{6}); // PHP Notice: Uninitialized string offset 6
?>
```

Any attempt to access a character before the beginning of the string (i.e., a negative offset) or after the end of the string will cause a runtime notice and the program will continue to run. One final error case happens when you attempt to add a character to an empty string:

```
<?php
$a = '';
$a{0} = 'x';
echo $a; // Array
?>
```

When you attempt to add a character to an empty string the string is silently (no messages) converted to an array whose value is the string '`Array`'.

PHP.8.2 Guidance to Language Users

- Although string access violations will not cause buffer overflows, they can cause unexpected behaviors so consider using bounds checking whenever using indexes from external sources.

PHP.9 Buffer Boundary Violation (Buffer Overflow) [HCB]

This vulnerability is not applicable to PHP because PHP's run-time checks the boundaries of arrays and causes an error when an attempt is made to access beyond a boundary.

PHP.10 Unchecked Array Indexing [XYZ]

This vulnerability is not applicable to PHP because PHP's run-time checks the boundaries of arrays and causes an error when an attempt is made to access beyond a boundary.

PHP.11 Unchecked Array Copying [XYW]

This vulnerability is not applicable to PHP because arrays are created, expanded, and contracted at run-time to the size and shape of the object being copied into them.

PHP.12 Pointer Casting and Pointer Type Changes [HFC]

This vulnerability is not applicable to PHP because PHP does not use pointers.

PHP.13 Pointer Arithmetic [RVG]

This vulnerability is not applicable to PHP because PHP does not use pointers.

PHP.14 Null Pointer Dereference [XYH]

This vulnerability is not applicable to PHP because PHP does not use pointers.

PHP.15 Dangling Reference to Heap [XYK]

This vulnerability is not applicable to PHP because PHP does not use pointers. Any reference to a deallocated variable causes a notice to be issued.

```
<?php
echo $x; // Issues a Notice: Undefined variable
$x = 1;
echo $x; // => 1
unset($x);
echo $x; // Issues a Notice: Undefined variable
?>
```

PHP.16 Arithmetic Wrap-around Error [FIF]

PHP.16.1 Applicability to Language

Wrap-around errors can occur whenever an attempt is made to in increase the value of a numeric type past its maximum value using arithmetic or shift operations (See PHP.17 Using Shift Operations for Multiplication and Division [PIK]for details about shift operations). If not detected at run-time and, dependent on the machine operational characteristics, the value can become a very small negative value which can cause a loop to run unexpectedly long generating unexpected results.

There are no exceptions thrown for wrap-around in PHP. This is not a direct problem for integers because increasing or decreasing an integer past its bounds will cause it to be interpreted as a float:

```
<?php
$a = PHP_INT_MAX;
echo var_dump($a);//int(2147483647)
$a = $a + 1;
echo var_dump($a);//float(2147483648)
?>
```

However, once an integer becomes a float then the behaviour of math using what may been expected to be an integer is no longer reliable.

PHP.16.2 Guidance to Language Users

- Be cognizant that arithmetic for integers that exceed their bounds becomes floating point math which may not have the exact same behaviour.
- Avoid using floating point or decimal variables for loop control but if you must use these types then bound the loop structures so as to not exceed the maximum or minimum possible values for the loop control variables.
- Test the implementation that you are using 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.

PHP.17 Using Shift Operations for Multiplication and Division [PIK]

PHP.17.1 Applicability to Language

Bit shifts are arithmetic: left shifts move zeroes in on the right with the sign bit lost, right shift preserve the sign bit.

```
<?php
printf("\nShift %d Right 1 bit:\n%08b\n%08b", -$x, -$x, -$x>>1);
printf("\nBitwise negation of %d:\n%032b\n%32b\n", $x, $x, ~$x);
?>
```

Executing the script above yields:

```
Shift -12 Right 1 bit:
11111111111111111111111111110100
11111111111111111111111111111010
Bitwise negation of 12:
00000000000000000000000000001100
11111111111111111111111111110011
```

PHP treats positive integers as being infinitely padded on the left with zeroes and negative numbers (in two's complement notation) with 1's on the left when used in bitwise operations:

```
$a<<$b // a shifted left b bits
$a>>$b // a shifted right b bits
```

The result of shifting left further than the number of digits in a value is unpredictable:

```
<?php
printf("100       =%032b", 100);
printf("\n100<<8    =%032b", 100<<8);
printf("\n100<<16   =%032b", 100<<16);
printf("\n100<<32   =%032b", 100<<32);
printf("\n100<<64   =%032b", 100<<64);
printf("\n100<<128  =%032b", 100<<128);
?>
```

The script above produces:

```
100       =00000000000000000000000001100100
100<<8    =00000000000000000110010000000000
100<<16   =00000000110010000000000000000000
100<<31   =00000000000000000000000000000000
100<<32   =00000000000000000000000001100100
100<<64   =00000000000000000000000001100100
```

There is no overflow check for shifting left or right so a program expecting an exception to halt it will instead unexpectedly continue leading to unexpected results. In the example above shifting left more than 31 places will leave the value unchanged with no warning that the operation failed. The same thing happens with shift right operations beyond 31 positions:

```
<?php
$x = pow(2,31);
printf("%d       =%032b", $x, $x);
printf("\n%d>>8    =%032b", $x, $x>>8);
printf("\n%d>>31   =%032b", $x, $x>>31);
printf("\n%d>>32   =%032b", $x, $x>>32);
?>
```

Executing the script above yields the original value:

```
-2147483648       =10000000000000000000000000000000
-2147483648>>8    =11111111100000000000000000000000
-2147483648>>31   =11111111111111111111111111111111
-2147483648>>32   =10000000000000000000000000000000
```

PHP.17.2 Guidance to Language Users

- Do not assume that you will get an over or underflow halt 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; and
- 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 (See PHP.52 Implementation–defined Behaviour [FAB] for more on this subject).

PHP.18 Sign Extension Error [XZI]

This vulnerability is not applicable to PHP because PHP converts between types without ever extending the sign.

PHP.19 Choice of Clear Names [NAI]

PHP.19.1 Applicability to Language

PHP uses the following rules to name variables, functions, constants, and classes:

- Names are of any length and consist of letters, numerals, underscores, and any character 127 through 255 (*0x7f-0xff*). Note that unlike some other languages where only the first *n* number of characters in a name are significant, ***all*** characters in a PHP name are significant. This eliminates a common source of name ambiguity when names are identical up to the significant length and vary afterwards which effectively makes all such names a reference to one common variable.
- All variable names must start with a dollar sign ($); and
- Variable names are case sensitive (e.g., `Alpha`, `ALPHA`, and `alpha` are each unique names). While this is a feature of the language that provides for more flexibility in naming, it is also can be a source of programmer errors when similar names are used which differ only in case (e.g., `aLpha` versus `alpha`).
- Function and class names are NOT case sensitive (e.g., `Alpha`, `ALPHA`, and `alpha` are each references to the SAME function or class).

PHP's naming rules are flexible by design but are also susceptible to a variety of unintentional coding errors:

- Variables are never declared but they must be assigned values before they are referenced. This means that some errors will never be exposed until runtime when the use of an unassigned variable will generate a notice (see also PHP.24 Initialization of Variables [LAV]).
- Variable names can be unique but may look similar to other names (e.g., `alpha` and `aLpha`, `__x` and `_x`, `_beta__` and `__beta_`) which could lead to the use of the wrong variable especially because PHP does not support declarations:

```
<?php
$veeeeeeeeerylongname = 'abc';
// do stuff
$veeeeeeeeeerylongname = 'xyz';//name has an extra 'e'
print($veeeeeeeeerylongname);//=> abc
?>
```

PHP utilizes dynamic typing with types determined at runtime. There are no type or variable declarations for a variable which can lead to subtle and potentially catastrophic errors:

```
<?php
$x = 1;
// lots of code…
```

```
    $X = 10;
?>
```

In the code above the programmer intended to set (lower case) $x to 10 and instead created a new *upper case* $X to 10 so the *lower case* $x remains unchanged. PHP will not detect a problem because there is no problem – it sees the upper case $X assignment as a legitimate way to bring a *new* object into existence.

PHP.19.2 Guidance to Language Users

- 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; and
- Use names that are clear and visually unambiguous.

PHP.20 Dead Store [WXQ]

PHP.20.1 Applicability to Language

It is possible to assign a value to a variable and never reference that variable which causes a "dead store." Other than the memory that this wastes, this normally is not very harmful, but if there is a substantial amount of dead stores then performance could suffer or, in an extreme case, the program could halt due to lack of memory. This could also provide unused space that could be exploited by attackers.

PHP.20.2 Guidance to Language Users

- Remove assignments to all variables that are never used.

PHP.21 Unused Variable [YZS]

The applicability to language and guidance to language users sections of the PHP.19 Dead Store [WXQ] write-up are applicable here.

PHP.22 Identifier Name Reuse [YOW]

PHP.22.1 Applicability to Language

Scoping allows for the definition of more than one variable with the same name to reference different objects. This can cause unpredicted behaviour if the reader of the code does not understand or notice the effects of scoping on variable assignment. For example:

```
<?php
$a = 1;
function x() {
```

```
  $a = 2;
}
x();
print "\$a = $a"; // $a = 1
?>
```

The `$a` variable within the function `x` above is local to the function only – it is created when `x` is called and disappears when control is returned to the calling program. If the function needed to update the outer variable named `$a` then it would need to specify that `$a` was a global before referencing it as in:

```
<?php
$a = 1;
function x() {
  global $a;
  $a = 2;
}
x();
print "\$a = $a"; // $a = 2
?>
```

In the case above, the function is updating the variable `$a` that is defined in the calling module.

Scoping rules cover other cases where an identically named variable name references different variables:

- A nested function's variables are in the scope of the nested function only; and
- Variables defined in outside a function are in global scope which means they are scoped to the program outside functions only and are therefore not visible within functions unless explicitly identified as `global` at the start of the function.

The concept of scoping makes it safer to code functions because the programmer is free to select any name in a function without worrying about accidentally selecting a name assigned to an outer scope which in turn could cause unwanted results. In PHP, one must be explicit when intending to circumvent the intrinsic scoping of variable names.

PHP.22.2 Guidance to Language Users

- Do not use identical names unless necessary to reference the correct object; and
- 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 make the code harder to understand.

PHP.23 Namespace Issues [BJL]

PHP.23.1 Applicability to Language

PHP has a hierarchy of namespaces which provides isolation to protect from name collisions and ways to

explicitly reference unique functions, constants, classes, and interfaces whose names could or would collide with other names. They also provide a way to create convenient (e.g., shorter) aliases for names. The rules for using them are complex and easily misunderstood which could lead to confusion:

```
1 <?php
2 namespace Alpha;
3 function f(){};
4 use X\Y as Z, A\B\C; // Importing for aliases
5 f();      // Calls function Alpha\f
6 \f();     // Calls function f defined in global scope
7 Z\f();    // Calls function X\Y\f
8 C\f();    // Calls function A\B\C\f
9 ?>
```

In the example above:

- Line 2 – The `namespace` statement defines the namespace to be used for the statements that follow it. Not that you can have more than one namespace defined serially;
- Line 4 – The `use` statement is used (at compile time only) to reference an externally defined qualified names. This is known as importing or aliasing. `X\Y as Z` means to substitute `X\Y` whenever `Z` is specified (see line 7 for example), `A\B\C` is a shorthand way to say "Use `A\B\C` whenever `C` is specified";
- Line 5 - Whenever there is no qualification the current namespace is used;
- Line 6 - Prepending with `\` specifies that the global space is to be used;
- Line 7 - `X\Y` substituted for `Z`; and
- Line 8 – `A\B\C` substituted for `C`.

In addition, you can define a function to be used at runtime to define missing classes/interfaces wheich is called when the runtime system is unable to resolve a reference.

PHP.23.2 Guidance to Language Users

- Use namespaces to differentiate your functions, constants, classes, and interfaces from global names and names within included files\; and
- Make certain you understand the rules that are used to avoid inadvertently referencing the wrong item.

PHP.24 Initialization of Variables [LAV]

PHP.24.1 Applicability of language

PHP does not check to see if a statement references an uninitialized variable until runtime. This is by design in order to support dynamic typing which in turn means there is no ability to declare a variable. PHP therefore has no way to know if a variable is referenced before or after an assignment. For example:

```
<?php
$b = 1;
if ($b > 0)
  echo "\$a = $a\n";// $a = [undefined variable]
  echo "Here"; // ==>Here
?>
```

The first `echo` statement is legal at compile time even if `$a` is not defined (i.e., assigned a value). A notice is raised at runtime only if the statement is executed and $b's current value is > 0. This scenario does not lend itself to static analysis because, as in the case above, it may be perfectly logical to not ever print `$a` unless `$b > 0`. Also note that only a notice is generated (i.e., no fatal error) and the program will continue to execute leading to unpredictable results.

PHP.24.2 Guidance to Language Users

- 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.

PHP.25 Operator Precedence/Order of Evaluation [JCW]

PHP.25.1 Applicability to Language

PHP provides many operators and levels of precedence so it is not unexpected that operator precedence and order of operation are not well understood and hence misused. For example:

```
<?php
echo 1 + 2 * 3, "\n"; // ==>7, evaluates as 1 + (2 * 3)
echo (1 + 2) * 3      // ==>9, parenthesis are allowed to coerce
precedence
?>
```

Be careful when using the assignment (`=`) operator:

```
<?php
$x = 1;
$y = null;
$z = isset($x)and isset($y);
echo '$x=', $x, ' $y=', $y, ' $z=', $z;// $x=1 $y =  $z=1
?>
```

In the example above, because `and` is higher precedence than `=`, the right side is evaluated first yielding `null` which results in:

```
$z = isset($x);
```

The resultant intermediate expression above evaluates to `true` which is almost certainly what not was

intended.

PHP.25.2 Guidance to Language Users

- Use parenthesis liberally to force intended precedence and increase readability;
- Be aware that PHP does not guarantee the order of evaluation for sub expressions; and
- Break large/complex statements into smaller ones using temporary variables for interim results.

PHP.26 Side-effects and Order of Evaluation [SAM]

PHP.26.1 Applicability to Language

Expressions that are evaluated left to right can cause a short circuit:

```
<?php
function a() {
  global $a;
  $a=1;
  return $a;
}
function b() {
  global $b;
  $b=2;
  return $b;
}
if (a()|| b()) {
  var_dump($a, $b); // Undefined variable b; prints int(1) NULL
}
?>
```

In the expression above function `b` is not evaluated because function `a` returns a TRUE value (i.e., 1) thus function `b` is never executed and `$b` is never defined. The use of the bitwise OR operator (`|`) instead of the logical OR operator (`||`), as above, results in the both functions being executed without any short circuiting.

PHP.26.2 Guidance to Language Users

- 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.

PHP.27 Likely Incorrect Expression [KOA]

PHP.27.1 Applicability to Language

Logical operators are `and`, `or`, `&&` and `||`; bitwise operators are `&` and `|`. Note that `and` and `or` have lower precedence than the `&&` and `||` symbols.

Testing for equivalence can be confused with assignment:

```
<?php
$a = 1;
if ($a==2) echo "\$a==2\n"; //False $a is not equal to 2
if ($a=2) echo "\$a=2"; //==> $a=2
?>
```

The first `if` statement is obviously `false` but the second `if` evaluate to `true` and it's not always obvious why. It's not because `2` is assigned to `$a` and then compared to it (no comparison takes place), it's because the assignment of `2` to `$a` returns a `true`.

Note in the example below that even though an assignment is made in the function parameter list it does not evaluate in the same manner as above:

```
<?php
function x($a) {
  echo "\$a=$a";
  return;
}
x($b=5);//==> $a=5
?>
```

The not logical operator (`!`) can cause confusion. In the example below, the not operator is applied to the first number – not the expression – therefore the `!$a==$b` test will always fail:

```
<?php
$a=1;
$b=2;
if (!$a==$b)
  echo '$a not equal to $b';
else
  echo '$a is equal to $b';//==> $a is equal to $b
?>
```

Popping the “last” element from an array does not delete the highest indexed element, it deletes the last *added*:

```
<?php
$a[1] = 'B';
$a[0] = 'A';
var_dump($a);
array_pop($a);
var_dump($a); [1]=> string(1) “B”
?>
```

PHP.27.2 Guidance to Language Users

- Move assignments outside of Boolean expressions;
- Do not confuse the equivalence operator (==) with the assignment operator (=);
- Bound not (!) operations with parenthesis [clarify];
- Simplify overly complex expressions; and
- Do not use assignment expressions is function parameter lists.

PHP.28 Dead and Deactivated Code [XYQ]

PHP.28.1 Applicability to Language

There are many ways to have dead or deactivated code occur in a program and PHP is no different than most other languages in that regard. Further, PHP does not provide static analysis to detect such code nor does the very dynamic design of PHP’s language lend itself to such analysis.

PHP.28.2 Guidance to Language Users

- Use static analysis tools to locate and remove dead-code.

PHP.29 Switch Statements and Static Analysis [CLL]

PHP.29.1 Applicability to Language

PHP provides a switch statement that provides a break, a default, and the ability to fall-through from one case to another as in the example below.

```
<?php
$a = 3;
switch($a) {
  case 1:
    echo "One";
  break;
  case 2:
    echo "Two";
```

```
    break;
  case 3:
    echo "Three"; //Fall-through
  default:
    echo "\nAll others";
}
?>
```

The code above will print:

```
Three
All others
```

This demonstrates the `default` statement as well as the ability to fall-through to other `case` statements when the `break` statement is not used.

PHP.29.2 Guidance to Language Users

- It's best to avoid fall-through from one case statement into the following case statement nut if necessary then provide a comment to inform the reader that the fall-through is intentional; and
- Generally speaking the `default` case should be used for handling all unexpected cases which should then be treated as errors.

PHP.30 Demarcation of Control Flow [EOJ]

PHP provides several ways of demarking control flow as illustrated below in the `if`/`else` statements. Other statements (e.g., `switch`) also share this ability to use alternate syntax:

```
<?php
$a=1;
if ($a > 0)   // Variation #1
  echo "\$a > 0\n";//==> $a > 0
else
  echo "\$a <= 0\n";

if ($a > 0) { // Variation #2 Using curly brace-enclosed block
  echo "\$a > 0\n";//==> $a > 0
} else {
  echo "\$a <= 0";
}

if ($a > 0) : // Variation #3: Using colon and end-if
  echo "\$a > 0";//==> $a > 0
else :
  echo "\$a <= 0";
endif;
```

```
?>
```

PHP.30.1 Applicability to Language

- 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; and
- Consider using the curly brace to demark blocks.

PHP.31 Loop Control Variables [TEX]

PHP.31.1 Applicability to Language

PHP provides two general loop control statements: `while` and `for`. It also provides a specialized loop control for arrays called `foreach`. These each support very flexible control constructs beyond a simple loop control variable.

PHP permits the modification of loop control variables within the body of the loop which can be overlooked which can lead to errors.

PHP.31.2 Guidance to Language Users

- Be careful to only modify loop control variables in ways that are easily understood and in ways that cannot lead to a premature exit or an endless loop.

PHP.32 Off-by-one Error [XZH]

PHP.32.1 Applicability to Language

The PHP language itself is vulnerable to off by one errors as is any language when used carelessly or by a person not familiar with PHP's index from zero versus from one. PHP does not prevent off by one errors but its runtime bounds checking for strings and arrays does lessen the chances that doing so will cause harm.

[Consider moving the example below since it's not really an off by one error]

It is possible to index past the end of a string by being off by one in which case PHP simply extends the length to accommodate the new character and pads with spaces:

```
<?php
$s = "abcdef";
$s[10] = "x";
echo "\n", $s;//==> abcdef    x
?>
```

PHP.32.2 Guidance to Language Users

- Be aware of PHP's indexing from zero and code accordingly.

PHP.33 Structured Programming [EWD]

PHP.33.1 Applicability to Language

PHP has only one statement which allows a program to be written in an explicitly unstructured manner- the `goto` statement. When used to branch out of a multilevel loop it can act as multi-level break:

```
<?php
for($i=0;$i<2; $i++) {
  for ($j=0; $j<2; $j++) {
    echo "\$i=$i \$j=$j\n";
    if (($i+$j) > 1) :
      goto there;
    else:
      continue;
    endif;
  }
  continue;
}
echo "not here";
there: echo "\nthere"
?>
```

In the example above the `goto` branches to `there` after 4 loops.

PHP does have one other statement that could be viewed as unstructured - the `break` statement. It's used in a loop to exit the loop and continue with the first statement that follows the last statement within the loop block. This is a type of branch but it is such a useful construct that few would consider it "unstructured" or a bad coding practice. It is arguably better than the `goto` which can easily be used to create unstructured code.

PHP.33.2 Guidance to Language Users

- 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; and
- Judicious use of `break` statements is encouraged to avoid confusion.

PHP.34 Passing Parameters and Return Values [CSJ]

PHP.34.1 Applicability to Language

PHP passes arguments by value by default but can also pass by reference by prepending the parameter with an ampersand (`&`):

```
<?php
$x=1;
$y=2;
```

```
function a($x, &$y) { // $x is passed by copy, $y by reference
  $x+=10;
  $y+=10;
  return;
}
a($x, $y);
echo "\$x=$x, \$y=$y";//==> $x=1, $y=12
?>
```

PHP.34.2 Guidance to Language Users

- 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;

PHP.35 Dangling References to Stack Frames [DCM]

This vulnerability is not applicable to PHP because, while PHP does provide a way to alter, or even inspect, data by address.

PHP.36 Subprogram Signature Mismatch [OTR]

PHP.36.1 Applicability to Language

PHP does not care if you pass all, some or none of the parameters expected in the argument list of the called function. Instead it provides three functions which allow the called function to logically interrogate the passed argument list:

```
<?php
function f($a, $b) {
  $n = func_num_args();
  if ($n > 1) {
    $args = func_get_args();
    for ($i=0; $i < $n; $i++) {
      echo "\$i=", $args[$i], " ";
    }
  }
  return;
}
f('A', 'B');//==>$i=A $i=B;
?>
```

PHP.36.2 Guidance to Language Users

- Though PHP supports variable argument lists it almost always clearer to have the function to specify explicit parameters and have the caller use those; and
- When a function is designed to operate on a variable number of items, consider using an array.

PHP.37 Recursion [GDL]

PHP.37.1 Applicability to Language

Recursion is supported in PHP but the manual recommends that no more 100-200 levels be utilized lest he stack overflow.

PHP.37.2 Guidance to Language Users

- Minimize the use of recursion and limit it to no more than 200 levels at most; and
- 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.

PHP.38 Ignored Error Status and Unhandled Exceptions [OYB]

PHP.38.1 Applicability to Language

PHP provides an error reporting function (`error_reporting`) which provides a runtime way to dynamically specify which PHP errors are to be reported. This can also be done using the php.ini file. PHP also provide numerous other non-OO functions that can be used to specify how to handle erros at runtime.

PHP also provides an exception class called `Exception` along with object-oriented statements (e.g., `throw`, `try`, and `catch`) to handle exceptions which considerably simplify the detection and handling of exceptions.

PHP.38.2 Guidance to Language Users

- Use PHP's exception handling with care in order to not catch errors that are intended for other exception handlers;
- 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;
- 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 etc.);
- 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; and
- Do not disable error checking;

PHP.39 Termination Strategy [REU]

PHP.39.1 Applicability to Language

PHP has a rich set of exception handling functions and classes which can be utilized to implement a termination strategy that assures the best possible outcome ranging from a hard stop to a clean-up and fail soft strategy. Refer to "PHP.38 Ignored Error Status and Unhandled Exceptions [OYB]

PHP.38.1 Applicability to Language

PHP provides an error reporting function (error_reporting) which provides a runtime way to dynamically specify which PHP errors are to be reported. This can also be done using the php.ini file. PHP also provide numerous other non-OO functions that can be used to specify how to handle erros at runtime.

PHP also provides an exception class called `Exception` along with object-oriented statements (e.g., `throw`, `try`, and `catch`) to handle exceptions which considerably simplify the detection and handling of exceptions.

PHP.38.2 Guidance to Language Users

- Use PHP's exception handling with care in order to not catch errors that are intended for other exception handlers;
- 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;
- 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 etc.);
- 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; and
- Do not disable error checking;

" for an example of an implementation that cleans up and continues.

PHP.39.2 Guidance to Language Users

- Use PHP's exception handling statements and/or `Exception` class to implement an appropriate termination strategy;
- Use PHP's numerous error detecting, reporting, and handling functions;
- Code with a three level approach: use coding practices which help prevent faults, detect, and handle faults;

PHP.40 Type-breaking Reinterpretation of Data [AMV]

This vulnerability is not applicable to PHP because the only way that two of more variables can reference the same storage area are through the use of references and references are not pointers and there is no way to have one or more references to a single storage area which do not all match in type.

PHP.41 Memory Leak [XYL]

PHP.41.1 Applicability to Language

PHP supports automatic garbage collection so in theory it should not have memory leaks. However, there are at least two general cases in which memory can be retained after it is no longer needed. The first is when implementation-dependent memory allocation/de-allocation algorithms (or even bugs) cause a leak – this is beyond the scope of this document. The second general case is when objects remain referenced after they are no longer needed. This is a logic error which requires the programmer to modify the code to delete references to objects when they are no longer required.

PHP.41.2 Guidance to Language Users

- Release all objects when they are no longer required.

PHP.42 Templates and Generics [SYM]

This vulnerability is not applicable to PHP because PHP does not implement these mechanisms.

PHP.43 Inheritance [RIP]

PHP.43.1 Applicability to Language

PHP supports multi-level inheritance, but not multiple inheritance (i.e., multiple levels of subclasses of a class are permitted but a subclass cannot inherit from more than one super class). Any inherited methods are subject to the same vulnerabilities that occur whenever using code that is not well understood.

PHP.43.2 Guidance to Language Users

- Document classes; and

- Inherit only from trusted classes.

PHP.44 Extra Intrinsics [LRM]

PHP.44.1 Applicability to Language

This vulnerability is not applicable to PHP because PHP does not provide mechanisms to override its internal built-in (i.e., intrinsic) functions.

PHP.45 Argument Passing to Library Functions [TRJ]

PHP.45.1 Applicability to Language

Refer to PHP.36 Subprogram Signature Mismatch [OTR].

PHP.45.2 Guidance to Language Users

Refer to PHP.36 Subprogram Signature Mismatch [OTR].

PHP.46 Inter-language Calling [DJS]

PHP.46.1 Applicability to Language

PHP.46.2 Guidance to Language Users

PHP.47 Dynamically-linked Code and Self-modifying Code [NYY]

PHP.47.1 Applicability to Language

PHP supports dynamic linking by design. The `include` statement …

This is the normal way in which external logic is made accessible to a PHP program therefore PHP is inherently exposed to any vulnerabilities that cause a different file to be imported:

- Alteration of a file directory path variable to cause the file search locate a different file first; and
- Overlaying of a file with an alternate.

PHP also provides an `eval`construct (not a function) which can be used to create self-modifying code:

```
<?php
$x = "echo 'Hello World';";
eval($x);//==> Hello World
?>
```

PHP.47.2 Guidance to Language Users

- Avoid using `eval` and *never* use with untrusted code;

PHP.48 Library Signature [NSQ] Need further investigation here

PHP.48.1 Applicability to Language

PHP has an extensive API for extending or embedding PHP using modules written in C. Extensions themselves have the potential for vulnerabilities exposed by the language used to code the extension which is beyond the scope of this document.

PHP.48.2 Guidance to Language Users

- Use only trusted modules as extensions; and
- If coding an extension utilize PHP's extension API to ensure a correct signature match.

PHP.49 Unanticipated Exceptions from Library Routines [HJW]

PHP.49.1 Applicability to Language

PHP is often extended by importing modules coded in PHP and other languages. For modules coded in PHP the risks include:

- Interception of an exception that was intended for a module's imported exception handling code (and vice versa); and
- Unintended results due to namespace collisions (covered in PHP.22 Identifier Name Reuse [YOW] and elsewhere in this document).

For modules coded in other languages the risks include:

- Unexpected termination of the program; and
- Unexpected side effects on the operating environment.

PHP.49.2 Guidance to Language Users

- Wrap calls to library routines and use exception handling logic to intercept and handle exceptions when practicable.

PHP.50 Pre-processor Directives [NMP]

This vulnerability is not applicable to PHP because PHP has no pre-processor directives.

PHP.51 Suppression of Run-time Checking [MXB]

PHP.52 Provision of Inherently Unsafe Operations [SKL]

PHP.53 Obscure Language Features [BRS]

PHP.53.1 Applicability of language

PHP has some obscure language features as described below:

Functions can be defined conditionally:

```
<?php
$a = 1;
// Assume $a's value could be 1 or not 1
if ($a == 1) {
  function f() {
    echo "\$a must equal 1";
  }
}
else {
    function f() {
      echo "\$a must not equal 1";
    }
  }
f();
?>
```

PHP.53.2 Guidance to Language Users

- 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;

PHP.54 Unspecified Behaviour [BQF]

PHP.54.1 Applicability of language

Though not documented PHP appears to convert strings that look like long integers into integers when comparing them:

```
<?php
$a = '999999999999999999991';
$b = '999999999999999999992';
If ($a == $b) echo "True\n"; //==> True
?>
```

PHP will convert an empty string to an array without warning when it's operated on:

```
<?php
$s = 'X';
$s[0] = 'Y';
echo $s."\n"; // ==> Y
$s[1] = 'Z';
echo $s."\n"; // ==> YZ
```

```
$s = '';
$s[0] = 'X';
echo $s."\n"; // => Array
?>
```

PHP's reference operator when used with an array element causes an apparent array copy to actually be a reference to the same location instead of a copied location:

```
<?php
$a1[0] = 1;
$a2 = $a1;
echo "\$a1[0]=$a1[0], \$a2[0]=$a2[0]\n";
$a2[0] = 2;
echo "\$a1[0]=$a1[0], \$a2[0]=$a2[0]\n";
//Now same thing but add a reference
$b1[0] = 1;
$x[0] =& $b1[0]; // Adding a reference changes the behaviour
$b2 = $b1;
echo "\$b1[0]=$b1[0], \$b2[0]=$b2[0], \$x[0]=$x[0]\n";
$b2[0] = 2;
echo "\$b1[0]=$b1[0], \$b2[0]=$b2[0], \$x[0]=$x[0]\n";
?>
```

Executing the above yields (minus *italics* and **bold underlined** text):

```
$a1[0]=1, $a2[0]=1   Arrays $a1 and $a2 are true copies
$a1[0]=1, $a2[0]=2
$b1[0]=1, $b2[0]=1, $x[0]=1 $b1 and $b2 are the same location
$b1[0]=2, $b2[0]=2, $x[0]=2
```

PHP.54.2 Guidance to Language Users

- Do not depend on the way PHP may or may not compare strings that contain long integers.

PHP.55 Undefined Behaviour [EWF]

PHP.55.1 Applicability to Language

PHP has undefined behaviour in the following instances:

- ```
  // mixing ++ and + produces undefined behavior
  $a = 1;
  echo ++$a + $a++; // may print 4 or 5
  ?>
  ```
- Automatic conversion of a string to an array
- Reusing a variable that is used as a reference
- Modulus with non integer numbers will give unpredictable results.
- Passing anything by reference other than a variable, new statement, or a return from a function.
  ```

- Converting to Integer from any type other than float, Boolean, or string.
- Converting to an integer from a float that is beyond the bounds of integers.

PHP.55.2 Guidance to Language Users

PHP.56 Implementation–defined Behaviour [FAB]

PHP.56.1 Applicability to Language

PHP has implementation-defined behaviour in the following instances:

- Need to document these

PHP.56.2 Guidance to Language Users

PHP.57 Deprecated Language Features [MEM]

PHP.57.1 Applicability to Language

PHP.57.2 Guidance to Language Users