ISO/IEC JTC 1/SC 22/WG 23 N 0331

Revised proposed Annex for Ruby Language

Date	2011-03-25
Contributed by	James Johnson
Original file name	
Notes	Replaces N0320

Annex Ruby

Ruby. Vulnerability descriptions for the language Ruby Standards and terminology

Ruby.1 Identification of standards and associated documents

IPA Ruby Standardization WG Draft – August 25, 2010

Ruby.2 General Terminology and Concepts

block: A procedure which is passed to a method invocation.

class: An object which defines the behaviour of a set of other objects called its instances.

class variable: A variable whose value is shared by all the instances of a class.

constant: A variable which is defined in a class or a module and is accessible both inside and outside the class or module. The value of a constant is ordinarily expected to remain unchanged during the execution of a program, but IPA Ruby Standardization Draft does not force it.

exception: An object which represents an exceptional event.

global variable: A variable which is accessible everywhere in a program.

implementation-defined: Possibly differing between implementations, but defined for every implementation.

instance method: A method which can be invoked on all the instances of a class.

instance variable: A variable that exists in a set of variable bindings which every object has.

local variable: A variable which is accessible only in a certain scope introduced by a program construct such as a method definition, a block, a class definition, a module definition, a singleton class definition, or the top level of a program.

method: A procedure which, when invoked on an object, performs a set of computations on the object.

method visibility: An attribute of a method which determines the conditions under which a method invocation is allowed.

module: An object which provides features to be included into a class or another module.

object: A computational entity which has states and behaviour. The behaviour of an object is a set of methods which can be invoked on the object.

singleton class: An object which can modify the behaviour of its associated object.

singleton method: An instance method of a singleton class.

unspecified behaviour: Possibly differing between implementations, and not necessarily defined for any particular implementation.

variable: A computational entity that refers to an object, which is called the value of the variable.

variable binding: An association between a variable and an object which is referred to by the variable.

Ruby.3 Type System [IHN]

Ruby.3.1 Applicability to language

Ruby employs a dynamic type system usually referred to as “duck typing”. In this system the class or type of an object is less important than the interface, or methods, it defines. Two different classes may respond to the same methods, i.e. instances of each class will handle the same method call. Usually an object is not implicitly changed into another type.

Automatic conversion occurs for some built-in types in certain situations. For example with the addition of a float and an integer, the integer will be converted automatically to a float. Note the result of an operation is indicated by a Ruby comment starting with =>.

```
a = 2

b = 2.0

a + b       #=> 4.0
```

Another instance of automatic conversion is when an integer becomes too large to fit within a machine word. On a 32-bit machine Ruby `Fixnums` have the range -2^{30} to $2^{30}-1$. When an integer becomes such that it no longer fits within said range it is converted to a `Bignum`. `Bignums` are arbitrary length integers bounded only by memory limitations.

Explicit conversion methods exist in Ruby to convert between types. The integer class contains the methods `to_s` and `to_f` which return the integer represented as a `string` object and `float` object, respectively.

```
10.to_s     #=> "10"

10.to_f     #=>  10.0
```

Strings likewise support conversion to integer and float objects.

```
"5".to_i    #=> 5

"5".to_f    #=> 5.0
```

Duck typing grants programmers of Ruby great flexibility. Strict typing is not imposed by the language, but if a programmer chooses, he or she can write programs such that methods mandate the class of the objects on which they operate. This is discouraged in Ruby. If an object is called with a method it does not know, an exception will be raised.

Ruby.3.2 Guidance to language users

- 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.
- Do not check for specific classes of objects unless there is good justification.

Ruby.4 Bit Representations [STR]

Ruby.4.1 Applicability to language

Ruby abstracts internal storage of integers. Users do not need to concern themselves about the size (in bits) of an integer. Since integers grow as needed the user does not need to worry about overflow. Ruby provides a mechanism to inspect specific bits of an integer through the `[]` method. For example to read the 10^{th} bit of a number:

```
number = 42

number[10] #=> 0

number = 1024

number[10] #=> 1
```

Note that the bits returned are not required to correspond to the internal representation of the number, just that it returns a consistent representation of the number in that implementation.

Ruby supports a variety of bitwise operators. These include ~ (not), & (and), | (or), ^ (exclusive or), << (shift left), and >> (shift right). Each of these operators works with integers of any size.

Ruby offers a pack method for the Array class (`Array#pack`) which produces a binary sequence dictated by the user supplied template. In this way members of an array can be converted to different bit representations. For instance an option for numbers is to store them in one of three ways: native endian, big-endian, and little endian. In this way bit sequences can be constructed for a particular interaction or purpose. There is a similar unpack method which will extract data given a template and bit sequence.

Ruby.4.2 Guidance to language users

- For values created within Ruby the user need not concern themselves with the internal representation of data. In most situations using specific binary representations makes code harder to read and understand.
- Network packets that go on the wire are one case where bit representation is important. In situations like this be sure to use the Array#pack to produce network endian data.
- Binary files are another situation where bit representation matters. The file format description should indicated big-endian or little endian preference.

Ruby.5 Floating-point Arithmetic [PLF]

Ruby.5.1 Applicability to language

Ruby supports the use of floating-point arithmetic with the Float class. The precision of floats in Ruby is implementation defined, however if the underlying system supports IEC 60559, the representation of floats shall be the 64-bit double format as specified in IEC 60559, 3.2.2.

Floating-point numbers are usually approximations of real numbers and as such some precision is lost. This is problematic when performing repeated operations. For example adding small values to numbers sometimes results in accumulation errors. Testing numbers for equality is sometimes unreliable as well. For this reason floating-point numbers should not be used to terminate loops.

Ruby.5.2 Guidance to language users

- Do not use a floating-point value in Boolean test for equality. Instead use code which determines if the number resides within an acceptable range.

Ruby.6 Enumerator Issues [CCB]

Ruby.6.1 Applicability to language

Ruby provides symbols for enumeration. Sometimes all which is required is to have unique representation, there is no value associated with the enumeration. In Ruby, symbols are lightweight objects which need not be defined ahead of time. For example,

```
travel(:north)
```

is a valid use of the symbol `:north`. (Ruby's literal syntax for symbols is a colon followed by a word.) There is no danger of accidentally getting to the "value" of an enumeration. So this:

```
travel(:north + :south)
```

is not allowed. Symbols do not support addition, or any method which alters the symbol.

Sometimes it is helpful to have values associated with enumerations. In Ruby this can be accomplished by using a hash. For example,

```
traffic_light = {
:green => "go"
:yellow => "caution"
:red => "stop"}


traffic_light[:yellow]
```

In this way values can be associated with the symbols. Members of a hash are accessed using the same bracket syntax as members of arrays. Note only integers can be used in array indexing, thus non-standard use of a symbol as an array index will raise an exception.

Ruby.6.2 Guidance to language users

- Use symbols for enumerators
- Do not define named constants to represent enumerators

Ruby.7 Numeric Conversion Errors [FLC]

Ruby.7.1 Applicability to language

Integers in the Ruby language are of unbounded length (the actual limit is dependent on the machine's memory). When an integer exceeds the word size for the machine there is no rollover and no errors occur. Instead Ruby converts the integer from one type to another. When possible, integers in Ruby are stored in a `Fixnum` object. `Fixnum` is a class which has limited integer range, yet is able to store the number efficiently in one machine word. Typically on a 32-bit machine the range is usually -2^{30} to $2^{30}-1$. These ranges are implementation defined.

Once calculations exceed this range, integers are stored in a Bignum object. Bignum class allows any length (memory providing) integer. This all takes place without the user's explicit instruction.

Ruby converts integers to floating point with the user's explicit intent. Loss of precision can occur converting from a large magnitude integer to a floating point number. This does not generate an error.

Ruby.7.2 Guidance to language users

- Have no concern for rollover errors or the magnitude of integers
- Enforce ranges on size dependent on the application

Ruby.8 String Termination [CJM]

This vulnerability is not applicable to Ruby since strings are not terminated by a special character

---

Ruby.9 Buffer Boundary Violation [HCB]

This vulnerability is not applicable to Ruby since array indexing is checked

---

Ruby.10 Unchecked Array Indexing [XYZ]

This vulnerability is not applicable to Ruby since array indexing is checked.

Ruby.11 Unchecked Array Copying [XYW]

This vulnerability is not applicable to Ruby since arrays grow.

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

This vulnerability is not applicable to Ruby since users cannot manipulate pointers.

Ruby.13 Pointer Arithmetic [RVG]

This vulnerability is not applicable to Ruby since users cannot manipulate pointers.

Ruby.14 Null Pointer Dereference [XYH]

This vulnerability is not applicable to Ruby since users cannot create or dereference null pointers.

Ruby.15 Dangling Reference to Heap [XYK]

This vulnerability is not applicable to Ruby since users cannot explicitly allocate and explicitly deallocate memory.

Ruby.16 Wrap-around Error [XYY]

This vulnerability is not applicable to Ruby since integers are unbounded.

Ruby.17 Sign Extension Error [XZI]

This vulnerability is not applicable to Ruby since users cannot explicitly convert a signed integer to a larger integer without modifying the value.

Ruby.18 Choice of Clear Names [NAI]

Ruby.18.1 Applicability to language

Ruby is susceptible to errors resulting from similar looking names. Ruby provides scoping of local variables. However, this can be confusing. Local variables cannot be accessed from another method, but local variables can be accessed from a block. Ruby features variable prefixes for non-local variables. The dollar sign signifies a global variable. A single "`@`" symbol signifies a variable scoped to the current object. A double at symbol signifies a class wide variable, accessible from any instance of said class.

Ruby.18.2 Guidance to language users

- Use names that are clear and visually unambiguous
- Be consistent in choosing names
- Use names which are rich in meaning
- Code will be reused in ways the original developers have not imagined

---

Ruby.19 Dead Store [WXQ]

Ruby.19.1 Applicability to language

Ruby is susceptible to errors of accidental assignments resulting from typos of variable names. Since variables do not need to declared before use such an assignment may go unnoticed. Such behaviour is indicative of programmer error.

Ruby.19.2 Guidance to language users

- Check that each assignment is made to the intended variable identifier
- Use static analysis tools, as they become available, to mechanically identify dead stores in the program

---

Ruby.20 Unused Variable [YZS]

This vulnerability is not applicable to Ruby variables cannot be declared.

Ruby.21 Identifier Name Reuse [YOW]

Ruby.21.1 Applicability to language

Ruby employs various levels of scope which allow users to name variables in different scopes with the same name. This can cause confusion in situations where the user is unaware of the scoping rules, especially in the use of blocks.

Modules provide a way to group methods and variables without the need for a class. To use these module and method names must be completely specified. For example:

```
Base64::encode(text)
```

However modules can be included, thus putting the contents of the module within the current scope. So:

```
include Base64

encode(text)
```

can cause clashes with names already in scope. When this occurs the current scope takes precedence, but the user may not realize this resulting in unknown errors.

Ruby.21.2 Guidance to language users

- Ensure that a definition does not occur in a scope where a different definition is accessible.
- Know what a module defines before including. If any definitions conflict, do not include the module, instead use the fully qualified name to refer to any definitions in the module.

Ruby.22 Namespace Issues [BJL]

Ruby.22.1 Applicability to language

This is indeed an issue for Ruby. The interpreter will resolve names to the most recent definition as the one to use, possibly redefining a variable. Scoping provides some means of protection, but there are

some cases where confusion arises. A method definition cannot access local variables defined outside of its scope, yet a block can access these variables. For example:

```
x = 50
def power(y)
puts x**y
end
power(2)   #=> NameError: undefined local variable or method 'x'
```

But the following can access the x variable as defined:

```
x = 50
def execute_block(y)
yield y
end
execute_block(2) {|y| x**y} #=> 2500
```

Ruby.22.2 Guidance to language users

- Avoid unnecessary includes
- Do not access variables outside of a block without justification

---

Ruby.23 Initialization of Variables [LAV]

This vulnerability is not applicable to Ruby since variables cannot be read before they are assigned.

Ruby.24 Operator Precedence/Order of Evaluation [JCW]

Ruby.24.1 Applicability to language

Ruby provides a rich set of operators containing over fifty operators and twenty levels of precedence. Confusion arises especially with operators which mean something similar, but are for different purposes. For example,

```
x = flag_a or flag_b
```

The above assigns the value of `flag_a` to `x`. If `flag_a` evaluates to false, then the value of the entire expression is `flag_b`. The intent of the programmer was most likely assign true to x if either `flag_a` or `flag_b` are true:

```
x = flag_a || flag_b
```

Ruby.24.2 Guidance to language users

- Use parenthesis around operators which are known to cause confusion and errors
- Break complex expressions into simpler ones, storing sub-expressions in variables as needed

---

Ruby.25 Side-effects and Order of Evaluation [SAM]

Ruby.25.1 Applicability to language

Ruby by definition strives on side-effects. Method invocations can change the state of the receiver (object whose method is invoked). This occurs not just for input and output for which side-effects are unavoidable, but also for routine operations such as mutating strings, modifying arrays, or defining methods. Ruby has adopted a naming convention which indicates destructive methods (those which modify the receiver) instead of creating a new object which is a modified copy. For example,

```
array = [1, 2, 3]      #=> [1, 2, 3]

array.slice(1..2)      #=> [2, 3]

array                  #=> [1, 2, 3]

array.slice!(1..2)     #=> [2, 3]
```

```
array                   #=> [1]
```

The method name with the exclamation signifies the object itself will be modified, whereas the other method does not modify it. Sometimes though the method is understood by the user to modify the object or cause side-effects. For example,

```
array = [1, 2, 3]

array.concat([4, 5, 6])

array #=> [1, 2, 3, 4, 5, 6]
```

These behaviours are documented and with little effort the user will be able recognize which methods cause side-effects and what those effects are.

The order of evaluation in Ruby is left to right. Order of evaluation and order of precedence are different. Precedence allows the familiar order of operations for expressions. For example,

```
a + b * c
```

`a` is evaluated, followed by `b` and `c`, then the value of `b` and the value of `c` are multiplied and added to the value of `a`. This is a subtle point which matters only if `a`, `b`, or `c` cause side effects. The following illustrates this:

```
def a; print "A"; 1; end

def b; print "B"; 2; end

def c; print "C"; 3; end

a + b * c  #=> 7, and "ABC" is printed to standard output
```

Ruby.25.2 Guidance to language users

- Read method documentation to be aware of side-effects
- Do not depend on side-effects of a term in the expression itself

---

Ruby.26 Likely Incorrect Expression [KOA]

Ruby.26.1 Applicability to language

Ruby has operators which are typographically similar, yet which have different meanings. The assignment operator and comparison operators are examples of these. Both are expressions and can be used in conditional expressions.

```
if a = 3 then #…

if a == 3 then #…
```

The first example assigns the value 3 to the variable a. 3 evaluates to true and the conditional is executed. The second checks that the variable `a` is equal to the value 3 and executes the conditional if true.

Another instance is the use of assignments in Boolean expressions. For instance,

```
a = x or b = y
```

This expression assigns the value `x` to `a`. If `x` is false then the value of y will be assigned to `b`. This should be avoided as the second assignment will not always occur. This could possibly be the intention of the programmer, but a more clear way to write the code which accomplishes that is:

```
a = x

b = y if a
```

There is no confusion here as the second assignment clearly has an if-modifier. This is common and well understood in the Ruby language.

Ruby.26.2 Guidance to language users

- Avoid assignments in conditions
- Do not perform assignments within Boolean expressions

Ruby.27 Dead and Deactivated Code [XYQ]

Ruby.27.1 Applicability to language

Dead and deactivated, as in any programming language with code branching, can be a problem in Ruby. The existence of code which can never be reached is not a problem itself. Its existence indicates the possibility of a coding error. Code coverage tools can help analyze which portions of code can and cannot be reached.

In particular the developer should ensure each branch can evaluate to true or false. If a condition only ever evaluates to true, then only one branch will be taken. This situation creates dead code.

Ruby.27.2 Guidance to language users

- Use analysis tools to identify unreachable code

Ruby.28 Switch Statements and Static Analysis [CLL]

Ruby.28.1 Applicability to language

Ruby provides a case statement. This construct is similar to C's switch statement with a few important differences. Cases do not "flow through" from one to the next. Only one case will be executed. An else case can be provided, but is not required. If no cases match then the value of the case statement is nil.

Ruby.28.2 Guidance to language users

- Include an else clause, unless the intention of cases not covered is to return the value nil
- Multiple expressions (separated by commas) may be served by the same when clause

---

Ruby.29 Demarcation of Control Flow [EOJ]

This vulnerability is not applicable to Ruby since control constructs require an explicit termination symbol.

Ruby.30 Loop Control Variables [TEX]

Ruby.30.1 Applicability to language

Ruby allows the modification of loop control variables from within the body of the loop. This is usually not performed, as the exact results are not always clear.

Ruby.30.2 Guidance to language users

- Do not modify loop control variables inside the loop body

Ruby.31 Off-by-one Error [XZH]

Ruby.31.1 Applicability to language

Like any programming language which supplies equality operators and array indexing, Ruby is vulnerable to off-by-one-errors. These errors occur when the developer creates an incorrect test for a number range or does not index arrays starting at zero.

Some looping constructs of the language alleviate the problem, but not all of them. For example this code

```
for i in 1..5
print i
end   #=> 12345
```

In addition to this is the usual confusion associated between <, <=, >, and >= in a test

Also unique to Ruby is the confusion of these particular loop constructs:

```
5.times {|x| p x}
```

and

```
1.upto(5) {|x| p x}
```

Each loop executes the code block five times. However the values passed to the block differ. With `5.times` the loop starts with the value `0` and the last value passed to the block is `4`. However in the case of `1.upto(5)`, it starts by passing `1`, and ends by passing `5`.

Ruby.31.2 Guidance to language users

- Use careful programming practice when programming border cases
- Use static analysis tools to detect off-by-one errors as they become available
- Instead of writing a loop to iterate all the elements of a container sue the `each` method supplied by the object's class

---

Ruby.32 Structured Programming [EWD]

Ruby.32.1 Applicability to language

Ruby makes structured programming easy for the user. Its object-oriented nature encourages at least a minimum amount of structure. However, it is still possible to write unstructured code. One feature which allows this is the break statement. The statement ends the execution of the current innermost loop. Excessive use of this may be confusing to others as it is not standard practice.

Ruby.32.2 Guidance to language users

While there are some cases where it might be necessary to use relatively unstructured programming methods, they should generally be avoided. The following ways help avoid the above named failures of structured programming:

- Instead of using multiple return statements, have a single return statement which returns a variable that has been assigned the desired return value
- In most cases a break statement can be avoided by using another looping construct. These are abundant in Ruby.
- Use classes and modules to partition functionality

Ruby.33 Passing Parameters and Return Values [CSJ]

Ruby.33.1 Applicability to language

Ruby uses call by reference. Each variable is a named reference to an object. Return values in Ruby are merely the object of the last expression, or a return statement. Note that Ruby allows multiple return values by way of array. The following is valid:

```
return angle, velocity      #=> [angle, velocity]
```

or less verbosely:

```
[angle, velocity]     #as the last line of the method
```

While pass by reference is a low over-head way of passing parameters, sometimes confusion can arise for programmers. If an object is modified by a method, then the possibility exists that the original object was modified. This may not the intended consequence. For example,

```
def pig_latin(word)
     word = word[1..-1] << word[0] if !word[/^[aeiouy]/]
     word << "ay"
end
```

The above method modifies the original object if it is that string starts with a vowel. The effect is the value outside the scope of the method is modified. The following revised method avoids this by calling the `dup` method on the object `word`:

```
def pig_latin_revised(word)
```

```
            word = word[/^[aeiouy]/] ? word.dup : word[1..-1] <<
word[0]

            word << "ay"

      end
```

Ruby.33.2 Guidance to language users

- Methods which modify their parameters should have the exclamation mark suffix. This is a standard Ruby idiom alerting users to the behaviour of the method
- Make local copies of parameters inside methods if they are not intended to be modified

---

Ruby.34 Dangling References to Stack Frames [DCM]

This vulnerability is not applicable to Ruby since users cannot create dangling references.

Ruby.35 Subprogram Signature Mismatch [OTR]

Ruby.35.1 Applicability to language

Subprogram signatures in Ruby only consist of an arity count and name. A mismatch in the number of parameters will thus be caught before a call is executed. The type of each parameter is not enforced by the interpreter. This is considered strength of Ruby, in that an object that responds to the same methods can imitate an object of another type. If an object does not respond to a method an error will be thrown. Also if the implementer chooses they can query the object to test its available methods and choose how to proceed.

Ruby.35.2 Guidance to language users

- The Ruby interpreter will provide error messages for instances of methods called with an inappropriate number of arguments

---

Ruby.36 Recursion [GDL]

Ruby.36.1 Applicability to language

Recursion can exhaust the finite stack space within a program. When this happens in Ruby, a "SystemStackError: stack level too deep" error occurs, which can be caught.

For methods which have the possibility of exhausting the stack, they should be implemented in an imperative style instead of the more mathematical, perhaps elegant, recursive manner.

There is no set amount of recursion an interpreter must support. Recursive methods which run successfully inside one conforming Ruby implementation may or may not successfully run inside a different implementation.

Ruby.36.2 Guidance to language users

- When possible, design algorithms in an imperative manner
- Test recursive methods extensively in the intended interpreter for stack overflow errors

---

Ruby.37 Returning Error Status [NZN]

Ruby.37.1 Applicability to language

Ruby provides the class Exception which is used to communicate between raise methods (methods which throw an exception) and rescue statements. Exception objects carry information about the exception including its type, possibly a descriptive string, and optional trace back.

Given this information the programmer can deal with exception appropriately within rescue statements. In some cases this might be program termination, while in other cases an error may be par for the course.

Ruby.37.2 Guidance to language users

- Extend Ruby's exception handling for your specific application
- Use the language's built-in mechanisms (`rescue`, `retry`) for dealing with errors

---

Ruby.38 Termination Strategy [REU]

Ruby.38.1 Applicability to language

Ruby standard does not explicitly state a termination strategy. The behaviour is unspecified. Differing implementations therefore can have different strategies.

Ruby.38.2 Guidance to language users

- Consult implementation documentation concerning termination strategy
- Do not assume each implementation behaves handles termination in the same manner

---

Ruby.39 Type-breaking Reinterpretation of Data [AMV]

This vulnerability is not applicable to Ruby since every data has a single interpretation.

Ruby.40 Memory Leak [XYL]

This vulnerability is no applicable to Ruby since users cannot explicitly allocate memory.

Ruby.41 Templates and Generics [SYM]

This vulnerability is not applicable to Ruby since it does not include templates or generics.

Ruby.42 Inheritance [RIP]

Ruby.42.1 Applicability to language

Ruby allows classes to inherit from one parent class. In addition to this modules can be included in a class. The class inherits the module's instance methods, class variables, and constants. Including modules can silently redefine methods or variables. Caution should be exercised when including modules for this reason. At most a class will have one direct superclass.

Ruby.42.2 Guidance to language users

- Provide documentation of encapsulated data, and how each method affects that data
- Inherit only from trusted sources, and, whenever possible check the version of the superclass during initialization
- Provide a method that provides versioning information for each class

Ruby.43 Extra Intrinsics [LRM]

This vulnerability is not applicable to Ruby since the most recent definition of a method is selected for use.

Ruby.44 Argument Passing to Library Functions [TRJ]

Ruby.44.1 Applicability to language

The original Ruby interpreter is written in the C language. Because of this many libraries for Ruby have been written to interface with the Ruby and C. The library designer should make the library validate any input before its use.

Ruby.44.2 Guidance to language users

- Develop wrappers around library functions that check the parameters before calling the function
- Use only libraries known to have been consistent and validated interface requirements

---

Ruby.45 Dynamically-linked Code and Self-modifying Code [NYY]

Ruby.45.1 Terminology and features

Dynamically-linked code might be a different version at runtime than what was tested during development. This may lead to unpredictable results. Self-modifying code can be written in Ruby.

Ruby.45.2 Description of vulnerability

- Verify dynamically linked code being used is the same as that which was tested
- Do not write self-modifying code

Ruby.46 Library Signature [NSQ]

Ruby.46.1 Terminology and features

Ruby implementations which interface with libraries must have correct signatures for functions. Creating correct signatures for a large library is cumbersome and should be avoided by using tools.

Ruby.46.2 Description of vulnerability

- Use tools to create signatures
- Avoid using libraries without proper signatures

Ruby.47 Unanticipated Exceptions from Library Routines [HJW]

Ruby.47.1 Terminology and features

Ruby interfaces with libraries which could encounter unanticipated exceptions. In some situations, largely dependent on the interpreter implementation, exceptions can cause unpredictable and possibly fatal results.

Ruby.47.2 Description of vulnerability

- Use library routines which specify all possible exceptions
- Use libraries which generate Ruby exceptions that can be `rescue`d

---

Ruby.48 Pre-processor Directives [NMP]

This vulnerability is not applicable to Ruby since it lacks a pre-processor.

Ruby.49 Obscure Language Features [BRS]

This vulnerability is not applicable to Ruby.

Ruby.50 Unspecified Behaviour [BQF]

Ruby.50.1 Applicability of language

Unspecified behaviour occurs where the proposed Ruby standard does not mandate a particular behaviour.

Unspecified behaviour in Ruby is abundant. In the proposed standard there are 136 instances of the phrase “unspecified behaviour.” Examples of

unspecified behaviour are:

- A `for`-expression terminated by a `break`-expression, `next`-expression, or `redo`-expression
- Calling `Numeric#coerce(numeric)` with the value `NaN`
- Calling `Integer#&(other)` if `other` is not an instance of the class `Integer`. This also applies to `Integer#|, Integer#^, Integer#<<`, and `Integer#>>`
- Calling `String#*(num)` if `other` is not an instance of the class `Integer`

Ruby.50.2 Guidance to language users

- Do not rely on unspecified behaviour because the behaviour can change at each instance.
- Code that makes assumptions about the unspecified behaviour should be replaced to make it less reliant on a particular installation and more portable.
- Document instances of use of unspecified behaviour

Ruby.51 Undefined Behaviour [EWF]

Ruby.51.1 Applicability to language

Undefined behaviour in Ruby is cover by sections [BQF] and [FAB].

Ruby.51.2 Guidance to language users

- Avoid using features of the language which are not specified to an exact behaviour.

Ruby.52 Implementation –defined Behaviour [FAB]

Ruby.52.1 Applicability to language

The proposed Ruby standard defines implementation-defined behaviour as: possibly differing between implementations, but defined for every implementation.

The proposed Ruby standard has documented 98 instances of implementation defined behaviour. Examples of implementation defined behaviour are:

- Whether a singleton class can have class variables or not
- The direct superclass of `Object`
- The visibility of **`Module#class_variable_get`**
- `Kernel.p(* args)` return value

Ruby.52.2 Guidance to language users

- The abundant nature of implementation-defined behaviour makes it difficult to avoid. As much as possible users should avoid implementation defined behaviour.
- Determine which implementation-defined implementations are shared between implementations. These are safer to use than behaviour which is different for every

---

Ruby.53 Deprecated Language Features [MEM]

This vulnerability is not applicable to Ruby since one edition of the standard exists.
