ISO/IEC JTC 1/SC 22/WG23 N1417

Deleted: N1416

Date: 2024-09-11

ISO/IEC WD 24772–4

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Programming languages — Avoiding vulnerabilities in programming languages – Part 4: Catalogue of vulnerabilities for the programming language Python

Commented [SM1]: For the ISO editor,
All code samples rely upon the spacing and arrangement of lines. Please, please do not touch them.

Document type: International standard

Document subtype: if applicable

Document stage: (10) development stage

Document language: E

Élément introductif — Élément principal — Partie n: Titre de la partie

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Participating in meeting 11 September 2024

Stephen Michell – convenor WG 23

Larry Wagoner - USA

Sean McDonagh – USA

Erhard Ploedereder – Germany

Tullio Vardanega – Italy

Regrets

Based on Document N 1415 based on N1408 28 August 2024

All issues discussed are captured in the document, either as comments or resolved issues.

Key for comments:

Deleted: with edits by Sean McDonagh.

Deleted: The previous reviewed version of this document is N1379....

Deleted: ¶ ... [1]

Contents

Deleted: 69

Deleted: 77

Deleted: 80
Deleted: 83
Deleted: 86
Deleted: 89
Deleted: 90
Deleted: 91
Deleted: 102
Deleted: 109

Deleted: 123

Deleted: 124

Deleted: 124

Deleted: 124

Deleted: 125

Deleted: 127

Foreword

ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces.

This document is part of a series of documents that describe how vulnerabilities arise in programming languages. ISO/IEC 24772-1:2024 addresses vulnerabilities that can arise in any programming language and hence is language independent. The other parts of the series are dedicated to individual languages.

This document provides avoidance mechanisms for vulnerabilities in the programming language Python, so that application developers considering Python or using Python will be better able to avoid the programming constructs that can lead to vulnerabilities in software written in the Python language and their attendant consequences. This document can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This document can also be used in comparison with companion documents and with the language-independent report, ISO/IEC 24772-1, "Programming Languages — Avoiding vulnerabilities in programming languages — Part 1:Language independent catalogue of vulnerabilities", to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

It should be noted that this document is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such document can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

1. Scope

This document specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, the avoidance mechanisms described herein are applicable to the software developed, reviewed, or maintained for any application.

This document describes how vulnerabilities specified in the language-independent ISO/IEC 24772–1 are manifested in Python.

Python is not an internationally specified language, in the sense that it does not have a single International Standard specification. The language definition is maintained by the Python Software Foundation at https://docs.python.org for the version of Python referenced in this document.

The analysis and avoidance mechanisms provided in this document are targeted to Python version 3.12 [15][16].

Implementations of earlier versions of Python exist and are in active usage, however, Python is not always backward compatible especially between v2.x and v3.x. Readers are cautioned to be aware of the differences as they apply to the avoidance mechanisms provided herein. To determine possible vulnerabilities for future releases of Python, research the documentation on the Python web site given above.

2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 24772-1:2023 Programming languages - Avoiding vulnerabilities in programming languages - Part 1: Language-independent catalogue of vulnerabilities

ISO/IEC 60559:2020 Information technology - Microprocessor Systems - Floating-Point arithmetic

ISO/IEC 10967-1:2012 Information technology - Language independent arithmetic - Part 1: Integer and floating-point arithmetic

ISO/IEC 10967-2:2001 Information technology - Language independent arithmetic - Part 2: Elementary numerical functions

ISO/IEC 10967-3:2006 Information technology - Language independent arithmetic - Part 3: Complex integer and floating-point arithmetic and complex elementary numerical functions

3. Terms and definitions

3.1 General

For the purposes of this document, the terms and definitions given in ISO/IEC 2382:2015, ISO/IEC 24772–1, and the following apply.

ISO and IEC maintain terminology databases for use in standardization are available at:

- IEC Glossary, std.iec.ch/glossary
- ISO Online Browsing Platform, www.iso.ch/obp/ui
- Python terminology can be found in the referenced Python document set https://docs.python.org.

3.2
annotation
label associated with a class or function name, variable or return value used as a type hint

3.3
argument
value passed to a function or method when called

3.4
assignment statement
statement that assigns an object to a name (label)

3.5
aware datetime object
objects that are aware of the time zone to which the object's value applies

3.6
body
the portion of a compound statement that follows the header and can contain other compound (nested) statements

3.7
Boolean
truth value where `True` corresponds to any non-zero value and `False` corresponds to zero

3.8
built-in
function provided by the Python language intrinsically without the need to import it (for example, `str()`, `slice()`, `type()`)

3.9
`class`
program-defined type which is used to instantiate objects and provide attributes that are common to all the objects that it instantiates

3.10
comment
information preceded by a "`#`" for readers and ignored by the language processor

3.11
complex number
number made up of a real and an imaginary part, each expressed as a floating-point number, in which the imaginary part is expressed with a trailing upper- or lower-case `j` or `J` or both

3.12
coroutine
generalized form of a subroutine used with `asyncio` that can be entered, exited, and resumed at many points

3.13
CPython
the standard implementation of Python coded in ANSI portable C

3.14
decorator
function that extends the behavior of another function without explicitly modifying it

3.15
dictionary
built-in mapping consisting of zero or more key:value pairs that are ordered, changeable, cannot contain duplicates, and can be indexed by keys of mixed types

3.16
docstring
one or more lines in a unit of code that are retrievable at run-time and serve to document the code

3.17
entry point
a mechanism for an installed distribution to offer specific execution services

3.18
exception
object that encapsulates the attributes of an error or abnormal event by terminating normal processing and can lead to program termination if not handled in the program

3.19
function
a grouping of statements, either built-in or defined in a program using the `def` statement, which can be called as a unit

3.20
garbage collection
process, controlled by the Python `gc` module, by which the memory used by unreferenced objects and their namespaces is reclaimed

Commented [SM2]: Renumber from here to the end of 3.

3.21
global object
object that is declared `global` and can be referenced from anywhere within the module or within any modules that import it

3.22
guerrilla patching
changing the attributes and/or methods of a module's class at run-time from outside of the module

3.23
Global interpreter lock (GIL)
mechanism in the CPython interpreter that limits execution to a single thread at a time

3.24
immutable object
object, such as an `int`, `float`, `bool`, `str`, or `tuple` object, whose value cannot be changed by the execution of the program

3.25
import
mechanism that is used to make the contents of a module accessible to the importing program

3.26
inheritance
definition of a `class` as a subclass of other classes such that inheriting class acquires methods and components from the superclass without explicitly defining them

3.27
instance
object that belongs to a `class` and created by invoking the `class` as if it were a function

3.28
integer
a whole number of any length

3.29
keyword
identifier that is reserved for special meaning to the Python interpreter and that cannot be used as a name of an object or a function or a method

3.30
lambda expression
an anonymous inline function consisting of a single expression which is evaluated when the function is called

3.31
list
ordered sequence of zero or more items which can be modified (mutable) and indexed

3.32
literal
string or number

3.33
membership
property of belonging by occurring in a sequence

3.34
method resolution order (MRO)
order used to resolve references to methods and variables to the correct inheritance level

3.35
module
file containing source code in Python or in another language and that has its own namespace and scope, may contain definitions for functions and classes, and is only executed once when first imported or reloaded

3.36
mutable
characteristic of being changeable such as a list or dictionary

3.37
naïve datetime object

objects that are not aware of the time zone to which the object's value applies

3.38
name
reference to a Python object such as a number, string, list, dictionary, tuple, set, built-in, module, function, or class

3.39
namespace
place where names reside with their references to the objects that they represent

3.40
None
null object

3.41
number
integer, floating-point, decimal, or complex number

3.42
operator
symbol that represents an action or operation on one or more operands

3. 43
overriding
attribute in a subclass to replace a superclass attribute

3.44
package
collection of one or more other modules in the form of a directory

3.45
pickling
process of serializing objects using the `pickle` module

3.46
scope
program region where a label or name is available for use within the overall program

3.47
script
unit of code generally synonymous with a program but usually run at the highest level

3.48
self
name of the class instance variable used within functions controlled by the instance

3.49
sequence
ordered container of mutable or immutable items of the same type that can be indexed or sliced using positive numbers

3.50
set
unordered sequence of zero or more mutable or immutable items which do not need to be of the same type

3.51
short-circuiting operator
logical operator consisting of two expressions where the evaluation of the right-hand expression can be skipped depending upon the operation and the evaluation of the left-hand expression

3.52
statement
any instruction written in the source code and executed by the Python interpreter

3.53
string
built-in immutable sequence object consisting of one or more characters and not containing a termination character

3.54
tuple
an immutable sequence of objects with potentially varying types

3.55
type hint
an annotation that identifies the expected type for a variable, `class`, function, or `return` value

3.56
variable
a reference to the memory location of an object that contains a value

4. Using this document

ISO/IEC 24772-1:2024 4.2 documents the process of creating software that is safe, secure and trusted within the context of the system in which it is fielded. As this document shows, vulnerabilities exist in the Python programming environment, and organizations are responsible for understanding and addressing the programming language issues that arise in the context of the real-world environment in which the program will be fielded.

Organizations following this document meet the requirements of 4.2 of ISO/IEC 24772-1, repeated here for the convenience of the reader:

- Identify and analyze weaknesses in the product or system, including systems, subsystems, modules, and individual components.
- Identify and analyze sources of programming errors.
- Determine acceptable programming paradigms and practices to avoid vulnerabilities using the documentation provided in 5.2, 6 and 7.
- Map the identified acceptable programming practices into organizational coding standards.
- Select and deploy tooling and processes to enforce coding rules or practices.

- Implement controls (in keeping with the requirements of the safety, security, and privacy needs of the system) that enforce these practices and procedures to ensure that the vulnerabilities do not affect the safety and security of the system under development.

In addition, organizations can determine avoidance and mitigation mechanisms using clause 6 of this document as well as other technical documentation, such as the MITRE Corporation, Common Weakness Enumeration (CWE) [8], Sun Microsystems, Inc. [18], and Einarsson [2]. Other views of avoiding programming mistakes and design flaws are addressed by Martelli [13] and Sebesta[17].

Tool vendors follow this document by providing tools that diagnose the vulnerabilities described in this document. Tool vendors also document to their users those vulnerabilities that cannot be diagnosed by the tool.

Programmers and software designers follow this document by following the architectural and coding guidelines of their organization, and by choosing appropriate mitigation techniques when a vulnerability is not avoidable.

5 General language concepts and primary avoidance mechanisms

5.1 General Python language concepts

5.1.1 Introduction

The key concepts discussed in this section are not entirely unique to Python, but they are implemented in Python in ways that are not always intuitive.

This document reflects material presented in the Python documentation set, which includes the Python Reference Manual [15] and the Python-C language interface [14]. Guidance regarding programming in Python can be found in Lutz [6] [7], Embedding Python [3], Python logging facility [5], Python runtime audit hooks [12] and packaging binary extensions [9].

5.1.2 Execution environment

All examples in this document were executed from the command line since an IDE (Integrated Development Environment) can optimize code and lead to different results.

5.1.3 Dynamic Typing

A frequent source of confusion is Python's dynamic typing and its effect on variable assignments (name is synonymous with variable in this annex). In Python there are no static declarations of variables. Variables are created, rebound, and deleted dynamically. Further, variables are not objects as they are in more traditional languages. Rather, they are references to objects and can be, and frequently are, bound to other objects as the program executes.

```
a = 1 # a is bound to an integer object whose value is 1
a = 'abc' # a is now bound to a string object
```

In Python, variables have no type – they reference objects which have types thus the statement `a = 1` creates a new variable called "`a`" that references a new object whose value is "`1`" and type is integer. That variable can be deleted with a `del` statement or bound to another object any time as shown above (see 6.2 Type system [IHN]). This annex often treats the term variable (or name) as being the object, which is technically incorrect but simpler. For example, in the statement `a = 1`, the object "`a`" is assigned the value "`1`". The name `a` is assigned to a newly created object of type integer which is assigned the value "`1`".

Even when explicit type declarations are present, they are not checked at runtime, and are instead checked using separate type checking tools. The following code will execute without any problems, but the assignment of a string to a variable explicitly declared as holding an integer will cause static type analysis to fail:

```
a: int = 1 # Programmer declares 'a' will always refer
           # to an int object
a = 'abc'  # Type checker reports error when a is bound
           # to 'a' string object
```

Similarly, there is no type checking for argument passing to user-defined functions and methods. Type errors are diagnosed during the execution of the function or method when an illegal operation is attempted, or a call is made to a function or method that is not defined. When type hints for function arguments are present, they can also be checked using separate type checking tools.

5.1.4 Mutable and Immutable Objects

Note that in the statement: `a = a + 1`, Python creates a new object whose value is calculated by adding `1` to the value of the current object referenced by `a`. If, prior to the execution of this statement `a`'s object had contained a value of `1`, then a new integer object with a value of `2` would be created. The integer object whose value was `1` is now marked for deletion using garbage collection provided no other variables reference it.

Note that the value of `a` is not updated in place, that is, the object referenced by `a` does not simply have `1` added to it as would be typical in other languages. This does not happen in Python because integer objects, as well as string, number and tuples, are immutable – they cannot be changed in place. Only lists, sets, and dictionaries can be changed in place – they are mutable. In practice this restriction of not being able to change a mutable object in place is mostly transparent but a notable exception is when immutable objects are passed as a parameter to a function or class (see 6.22 Initialization of Variables [LAV]).

The underlying actions that are performed to enable the apparent in-place change do not update the immutable object – they create a new object and bind (or "point") the variable to the new object. This can be shown as below (the id function returns an object's address):

```
a = 'abc'
print(id(a)) #=> 30753768
a = 'abc' + 'def'
print(id(a)) #=> 52499320
print(a) #=> abcdef
```

The updating of objects referenced in the parameters passed to a function or class is governed by whether the object is mutable, in which case it is updated in place, or immutable in which case a local copy of the object is created and updated which has no effect on the passed object. This is described in more detail in 6.32 Passing Parameters and Return Values [CSJ].

5.1.5 Variables, objects, and their values

Python variables (names) are not like variables in most other languages - they are dynamically referenced to objects. Python allows optional explicit type declarations to be added to variables, function parameters and return values. The Python language itself does not enforce these annotations but they can be used by third-party type checkers, as well as IDEs. Any Python variable can be reassigned to objects of different types at different times.

Python creates each variable when it is first assigned. In fact, assignment is the only way to bring a variable into existence. Function parameters are implicitly assigned by the interpreter when the function is called. All values in a Python program are accessed through a variable reference which points to a memory location which is always an object (for example, number, string, list, and so on). A variable is said to be bound to

an object when it is assigned to that object. A variable can be rebound to another object which can be of any type. For example:

```
a = 'alpha' # assignment to a string
a = 3.142 # rebinding "a" to a float
a = b = (1, 7.4, "Hello") # rebinding to a tuple
print(a) #=> (1, 7.4, "Hello")
del a
print(b) #=> (1, 7.4, "Hello")
print(a) #=> NameError: name 'a' is not defined
```

The first three statements show dynamic binding in action. The variable a is bound to a string, then to a float, then to another variable which in turn is assigned a tuple of value (1, 7.4, "Hello"). Tuples can contain objects of mixed types and are immutable and ordered.

The `del` statement then unbinds the variable a from the tuple object which effectively deletes the variable `a` (if there were no other references to the tuple object it too would have been deleted because an object with zero references is marked for garbage collection (but is not necessarily deleted immediately)). In this case, we see that b is still referencing the tuple object, so the tuple is not deleted. The final statement above shows that an exception is raised when an unbound variable is referenced.

The way in which Python dynamically binds and rebinds variables is a source of some confusion to new programmers and even experienced programmers who are used to static binding where a variable is permanently bound to a single memory location. Values are assigned to objects which in turn are referenced by variables but it is simpler to say the value is assigned to the variable. For brevity this document uses this simpler, though not as exact, wording. Variables in an expression are replaced with object references when that expression is evaluated, therefore a variable must be explicitly assigned before being referenced, otherwise a run-time exception is raised:

```
a = 1
if a == 1 : print(b) # error – b is not defined
```

When line 1 above is interpreted, an object of type integer is created to hold the value `1` and the variable `a` is created and linked to that object. The second line illustrates how an error is raised if a variable (`b` in this case) is referenced before being assigned to an object.

```
a = 1
b = a
a = 'x'
print(a, b) #=> x 1
```

Variables can share references as above – `b` is assigned to the same object as `a`. This is known as a shared reference. If `a` is later reassigned to another object (as in line 3 above), `b` will still be assigned to the initial object that a was assigned to when `b` shared the reference, in this case `b` would equal to `1`.

The subject of shared references requires particular care since its effect varies according to the rules for in-place object changes. In-places object changes are allowed only for mutable (that is, alterable) objects. Numeric objects and strings are immutable (unalterable). Lists and dictionaries are mutable which affects how shared references operate as below:

```
a = [1,2,3]
b = a
a[0] = 7
print(a) # [7, 2, 3]
print(b) # [7, 2, 3]
```

In the example above, `a` and `b` have a shared reference to the same list object so a change to that list object affects both references. If the shared reference effects are not well understood, the change to `b` can cause unexpected results.

Assignments can also invoke an augmented syntax such as `a += 1`. Other syntaxes support multiple targets, that is,

```
x = y = z = 1
```

binding (or rebinding) an instance attribute, that is,

```
x.a = 1
```

and binding (or rebinding) a container element, that is,

```
x[k] = 1
```

For further discussion of aliasing see 6.32 Passing parameters and return values [CSJ], and 6.38 Deep vs shallow copying [YAN]. For further discussion of concurrent access to values, see 6.61 Concurrency - data access [CGX].

The Python language, by design, allows for dynamic binding and rebinding. Because of the dynamic way in which variables are brought into a program at run-time, the Python

language runtimes cannot warn that a variable is referenced but never assigned a value. The following code illustrates this:

```
if a > b:
    import x
else:
    import y
```

Depending on the current value of `a` and `b`, either module `x` or `y` is imported into the program. If `x` assigns a value to a variable `z` and module `y` references `z` then dependent on which `import` statement is executed first (an import always executes all code in the module when it is first imported), an unassigned variable reference exception will or will not be raised.

Programmers can use `ResourceWarning` to detect the implicit cleanup of resources and `tracemalloc` to report the location of the resource allocation.

Python only checks whether a variable already exists when it is encountered in a statement that attempts to access its value. It was intentionally part of the Python language design to resolve names at runtime when they are used. This allows for the scoping semantics where names may be resolved in either the current local scope, an outer lexically nested function scope, the module global, or the built-in namespace. Python therefore has no way to know if a variable is referenced before or after an assignment. For example:

```
if y > 0:
    print(x)
```

The above statement is legal even if `x` has not been previously defined (that is, assigned a value) in the current scope or an outer lexically nested function scope in a way that is visible to the compiler. However, at runtime, an exception `UnboundLocalError` is raised when a local variable is read before it is assigned. The exception is raised only if the statement is executed and `y > 0`, and `x` is not present in the current local scope, module globals or the built-in namespace. Thus, this scenario would not lend itself to static analysis because, as in the case above, it may be perfectly logical to not ever print `x` unless `y > 0`, or the program may use means that are opaque to the compiler to ensure that x is available in the module scope or the built-in namespace by the time it is needed (for example, it may be set from another module, or programmatically via the `globals()` built-in).

There is no ability to use a variable with an uninitialized value because assigned variables always reference objects which always have a value and unassigned variables do not exist. Therefore, Python raises an exception at runtime when an unassigned (that is, non-existent) variable is referenced.

Initialization of function arguments can cause unexpected results when an argument is set to a default object which is mutable:

```
def x(y=[]):
    y.append(1)
    print(y)
x([2]) #=> [2, 1], as expected (default was not needed)
x() # [1]
x() # [1, 1] continues to expand with each subsequent call
```

The behaviour above is not a bug, it is a defined behaviour for mutable objects, but it is a very bad idea in almost all cases to assign mutable objects as default values.

5.1.6 Inheritance

Inheritance is a powerful part of Object-Oriented Programming (OOP). Python supports single inheritance and multiple inheritance.

Python supports inheritance through a dynamic hierarchical search of class namespaces starting at the class of a given object and proceeding upward through its superclasses. Python supports method overriding; it does not support method overloading by default.

In binding the name of a method call, normally only the name of the called function is considered. As a special case, the decorator `@dispatch` can be used to enable method overloading, but only within the namespace of a single class. The decorator does not allow for overloading of methods along an inheritance hierarchy. Consider:

```
from multipledispatch import dispatch

@dispatch(int,int)
def product(first, second):
    result = first*second
    print(result)

@dispatch(float,float,float)
def product(first, second, third):
    result  = first * second * third
    print(result)

product(2,3) # => 6
```

```
product(2.2,3.4,2.3) # => 17.204
```

Without the `@dispatch` decorators, only the second method product would be considered in subsequent name binding. With the decorators, the types of the parameters are taken into account as well in binding the method name of a call.

As the name resolution takes only the method name into account, a method definition either redefines (hides) an equally named inherited method of the class of the object or, if `none` is found, it represents a new method.

```
class A:
    def method1(self):
        print('method1 of class A')

class B(A):
    def method1(self):
        print('Modified method1 of class A by class B')

b = B()
b.method1() #=> Modified method1 of class A by class B
```

Multiple inheritance is also supported by Python. Name resolution uses a strategy known as Method Resolution Order (MRO). The MRO is also commonly recognized as C3 Linearization. For simpler cases that do not involve “diamond structures” (i.e., superclasses that are shared by other superclasses), the MRO generally follows a depth-first, left-to-right ordering protocol resulting in a single path through the inheritance tree. For diamond structures, the rules become more complicated as shown by the examples below. In these cases, the MRO is difficult to establish manually, and its outcome differs substantially from the usual rules in other OO-languages. In general, the MRO lookup sequence for binding names in classes is a mixture of left-most depth-first and selective breadth-first traversal, the latter ensuring that all search paths back to a given parent node are explored before this parent node is visited.

Consider the following example of multiple inheritance:

```
class A:
    def __init__(self):
        self.id = 'Class A'
    def getId(self):
        return "from A " + self.id

class B:
    def __init__(self):
        self.id = 'Class B'
    def getId(self):
```

```
        return "from B " + self.id

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)

c = C()
print(c.getId()) # => from A Class B
                 # When class C(B,A) is used,
                 # the output is -> from B Class B
```

Even though both class A and class B carry a component id, the joint child C class has a single instance of id. Thus, the assignments executed by `A.__init__(self)` and `B.__init__(self)` operate on this single instance overwriting each other.

The built-in function `super()` introduces more flexibility. In Python, `super()` also relies on MRO. Updating the previous example using `super()` is shown below and the output is now `class A`. Reversing the inheritance call to class `C(B, A)` would predictably result in `class B`. The MRO for the scenario below is calculated using the `__mro__` attribute for `class C` resulting in (`C -> A -> B`). It is important to make sure that each class calls the `__init__` of its superclass so that it is properly initialized.

```
class A:
    def __init__(self):
        super().__init__()
        self.id = 'Class A'
    def getId(self):
        return self.id

class B:
    def __init__(self):
        super().__init__()
        self.id = 'Class B '
    def getId(self):
        return self.id

class C(A, B):
    def __init__(self):
```

```
        super().__init__()
    def getId(self):
        return self.id

c = C()
print(c.getId()) # => Class A
print(C.__mro__) # => (<class '__main__.C'>,
                 # <class '__main__.A'>, <class
'__main__.B'>,
                 # <class 'object'>)
```

In general, the MRO lookup sequence for binding names in classes is a mixture of left-most depth-first and selective breadth-first traversal; the latter ensuring that all search paths back to a given parent node are explored before this parent node is visited. As noted earlier, in these cases the MRO is difficult to establish manually. Additionally, Python renders certain MRO's illegal which further complicates the understanding of the rules. For example, in a class hierarchy described by

```
class O: pass
class P: pass
class A(P): pass
class B(P): pass
class Z(O): pass
class Y(Z): pass
class W(O): pass

class C(Y, A, B, W): pass # This works fine

c = C()
c.meth()

class C(Z, Y, A, B, W): pass
                    # => TypeError: Cannot create a
                    # consistent MRO for bases Z, Y, A, B, W
```

the MRO for resolving the method name `c.meth()` is the linear sequence

```
C - Y - Z - A - B - P - W - O - object.
```

On the other hand, in the last line above, Python cannot establish a consistent MRO for

```
class C(Z, Y, A, B, W),
```

because `Z` is a superclass of `Y` and Python throws the `TypeError` exception. Notice that `object` is always the last class in every MRO chain.

Note that Python will always diagnose a failure to declare a legal class, as shown above.

5.1.7 Concurrency

Python's threading module provides the ability to perform cooperative multithreading from within a single native thread. Due to the restrictions of Python's Global Interpreter Lock (GIL) in some implementations, only one thread at a time is permitted to run. In these implementations, multithreading can still be useful in situations where the CPU becomes idle such as in I/O-bound applications.

It is important to handle potential thread exceptions when starting new threads, and care needs to be taken so that each thread is only started once.

Python's multiprocessing module provides multiprocessing capability that allows independent processes to run on multiple cores. Unlike threading, these independent processes do not have shared memory and are not prone to the relevant data races. It is important to handle potential multiprocessing exceptions when starting new processes, and if a process terminates as the result of an exception, it cannot be restarted.

Python's asyncio module is the newest approach to handling asynchronous concurrency, introduced in Python 3.4. This new asyncio processing model is typically faster than implementations that use traditional threads and multiprocessing, and it is often safer since asyncio operations all run in the same thread. Python event loops are automatically generated by `asyncio.run()`. When using asyncio, correct operation requires that all tasks relinquish control co-operatively, with execution controlled by the Async IO manager. Since task switching is less arbitrary than thread context switching when cooperative transfers of control between coroutines are used, i.e., `await()` to provide predictable control over the task switching process. Multiple event loops are possible but not recommended when using `asyncio` as the execution model relies on a single thread and adding multiple event loops does not provide additional functionality or performance.

Interprocess communication is almost always necessary in multicore systems. If a program is implemented that uses both processes and event loops, it should be noted that event loops are not a suitable means of interprocess communication. It is

plausible, however, that the masters of event loops may need to communicate with one another, and this should happen outside of the event loop processing.

A thread with the `daemon` flag set to `True` is called a daemon thread and never terminates until the program ends.

Futures are Python objects that represent the eventual result of asynchronous operations. Futures are also available using the `concurrent.futures` module, which provides a common interface for asynchronous execution of threads using `ThreadPoolExecutor`, or processes using `ProcessPoolExecutor`. When executors are used, the overheads of repeatedly creating threads or processes are avoided. For CPU bound tasks, the `ProcessPoolExecutor` class can provide better performance. Futures in `asyncio` are awaitable objects and are not thread safe. Coroutines await on future objects until they provide a valid result, error message, or are cancelled.

5.2 Primary avoidance mechanisms for Python

5.2.1 Recommendations in interpreting ISO/IEC 24772-1 avoidance mechanisms

Python has some fundamental differences with standard imperative languages, which are the majority of languages covered by the ISO/IEC series of documents, and the avoidance mechanisms offered by those documents do not always apply to Python. This document describes how the vulnerabilities identified in ISO/IEC 24772-1 manifest in Python and the steps recommended to mitigate them.

The expectation is that users of this document will develop and use a coding standard based on this document that is tailored to their risk environment.

5.2.2 Top avoidance mechanisms

Each vulnerability listed in clause 6 provides a set of ways that the vulnerability can be avoided or mitigated. Many of the mitigations and avoidance mechanisms are common. This clause provides the most effective and most common mitigations, together with references to which vulnerabilities they apply. The references to the respective vulnerabilities are provided to give the reader easy access to those vulnerabilities for rationale and further exploration. The mitigations provided here are in addition to the top avoidance mechanisms provided in ISO/IEC 24772-1:2024 5.4.

TABLE 1: Top avoidance mechanisms in Python

Number	Recommended avoidance mechanism	Applicable vulnerabilities
1	Use type annotations to help provide static type checking prior to running code.	6.5 [CCB] 6.2 [IHN] 6.11 [HFC] 6.41 [RIP] 6.42 [BLP] 6.44 [BKK]
2	Avoid the use of `pickle`, but if it must be used, only unpickle trusted data.	6.53 [SKL] 6.61 [CGX]
3	Avoid implicit references to global values from within functions to make code clearer. In order to update `global` objects within a function or class, place the `global` statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are global (for example, `global a, b, c`).	6.20 [YOW] 6.21 [BJL] 6.61 [CGX] 6.63 [CGM]

Number	Recommended avoidance mechanism	Applicable vulnerabilities
4	Always use named exceptions to avoid catching errors that are intended for other exception handlers and use context managers to enclose the code creating the exception.	6.6 [FLC] 6.15 [FIF] 6.31 [EWD] 6.36 [OYB] 6.59 [CGA] 6.62 [CGS]
5	Avoid using `exec` or `eval` and never use these with untrusted code.	6.48 [NYY] 6.53 [SKL]
6	Avoid guerrilla patching, but if unavoidable, be aware that altering the behavior of objects at runtime can make code much more difficult to understand and can introduce vulnerabilities.	6.48 [NYY] 6.53 [SKL]
7	Consider the guidance of “PEP 551 – Security transparency in the Python runtime” [11] and “PEP 578 Python Runtime Audit Hooks" [12] when using audit hooks.	6.48 [NYY] 6.54 [BRS]
8	Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap-around errors.	6.15 [FIF]
9	When using multiple threads, verify that all shared data is protected by locks or similar mechanisms, and use inter-communication mechanisms or global references to ensure safe terminations.	6.59 [CGA] 6.60 [CGT] 6.61 [CGX] 6.63 [CGM]

Number	Recommended avoidance mechanism	Applicable vulnerabilities
10	Avoid mixing concurrency models within the same program.	6.59 [CGA]
11	When using `asyncio`, make all tasks non-blocking.	6.25 [KOA] 6.59 [CGA] 6.61 [CGX] 6.65 [BQF]
12	Avoid external termination of concurrent entities except as an extreme measure.	6.60 [CGT]
13	Be cognizant of the precise semantics of assignments to mutable objects.	6.54 [BRS]
14	Inherit only from trusted classes and only use multiple inheritance that is linearizable with the MRO rules.	6.41 [RIP] 6.43 [PPH]
15	Avoid logic that depends on byte order or use the `sys.byteorder` variable and write the logic to account for byte order.	6.57 [FAB] 6.3 [STR]

6. Programming language vulnerabilities in Python

6.1 General

Clause 6 contains specific analysis for the Python programming language about the possible presence of vulnerabilities as described in ISO/IEC 24772-1:2024 and provides specific avoidance mechanisms for Python. This section mirrors ISO/IEC

24772-1:2024 Clause 6 in that the vulnerability "Type system [IHN]" is found in 6.2 of ISO/IEC 24772-1:2024, and Python specific avoidance mechanisms are found in 6.2 "Type system [IHN]" in this document.

Note that the avoidance mechanisms provided in this document apply to Python as specified in the Python 3.12 documentation. Python is extended by several commonly used libraries that can have behaviours different from those documented by the Python standard. This document does not address these additional libraries.

Commented [SJM3]: The latest version is now v 3.12.5.

6.2 Type system [IHN]

6.2.1 Applicability to language

The vulnerabilities related to insufficient use of the type system as specified in ISO/IEC 24772-1:2024 6.2 apply to Python.

Python abstracts all data as objects and every object has a type (in addition to an identity and a value). Extensions to Python, written in other languages, can define new types, and Python code can also define new types, either programmatically through the `types` module, or by using the dedicated `class` statement.

Python is also a strongly typed language – operations cannot be performed on an object that is not valid for that type. Checks performed to ensure an appropriate type are performed dynamically when the operation on the object is invoked. For operations that are not valid for a given type, an exception will be raised at runtime. Programmers can use `isinstance()`, `type()`, and other behavioural based type checkers to verify that the type is valid or convertible, and then convert to the desired type.

```
a = 'abc' # a refers to a string object
if isinstance(a, str): print('a type is string')
```

By default, a Python program is free to assign (bind), and reassign (rebind), any variable to any type of object at any time. This is considered safe in general since the type of the object is carried in the object and if a variable is rebound, then any future calls using that variable will check the type recorded in the object to decide the validity of the operation. See 6.36 Ignored error status and unhandled exceptions [OYB] for a discussion of the vulnerabilities associated with failed checks.

In Python, variables are created when they are first assigned a value (see 6.17 Choice of clear names [NAI]). Variables are generic in that they do not have a type. They simply reference objects which hold the object's type information.

Automatic conversion occurs only for numeric types of objects. Python converts (coerces) from the simplest type up to the most complex type whenever different numeric types are mixed in an expression. For example:

```
a = 1
b = 2.0
c = a + b; print(c) #=> 3.0
```

In the example above, the + operation converts the value of a to its floating-point equivalent, 1.0, adds it to b, and stores the floating-point value, 3.0, into c (which is thus a floating-point number). A programmer may erroneously expect that c is an integer and use it accordingly which can lead to unexpected results.

Some of these issues are visible to the programmer. For example, x = 1/2 will create an object of type float with a numeric value of 0.5, while x = 1//2 will truncate to the integer 0.

Gradual typing in Python allows optional annotations to be added to dynamic variables to assign them types so that they can be statically checked. This lets Python programs contain both dynamic variables, while adding the error-checking benefits of statically typed variables. Python tools provide static type checkers that assist users in avoiding the misuse of declared types in Python.

Python also has the vulnerability that changes in logical representation (e.g., meters to feet) are not enforced by the general type system. Programmers can use dedicated libraries to manage such types or can create their own using classes.

Commented [SJM4]: Reword? Consider: ...that *changes* of logical representation...

6.2.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Python software developers can:

- Follow the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.2.5.
- Use static type checkers to detect typing errors. The Python community is one source of static type checkers.
- Pay special attention to issues of magnitude and precision when using mixed type expressions.
- Be aware of the consequences of shared references (see 6.24 Side-effects and order of evaluation of operands [SAM] and 6.38 Deep vs. shallow copying [YAN]).
- Keep in mind that using a very large integer will have a negative effect on performance.

6.3 Bit representations [STR]

6.3.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.3 apply to Python.

Python provides hexadecimal, octal and binary built-in functions. `oct` converts to octal, `hex` to hexadecimal and `bin` to binary:

```
print(oct(256)) # 0o400
print(hex(256)) # 0x100
print(bin(256)) # 0b100000000
```

The notations shown as comments above are also valid ways to specify octal, hex and binary values respectively:

```
print(0o400) #=> 256
a = 0x100+1; print(a) #=> 257
```

The built-in `int` function can be used to convert strings to numbers and optionally specify any number base:

```
int('256') # the integer 256 in the default base 10
int('400', 8) #=> 256
int('100', 16) #=> 256
int('24', 5) #=> 14
```

Python stores integers that are beyond the underlying hardware's largest integer size as an internal value of arbitrary length so that programmers are only limited by performance concerns when very large integers are used (and by memory when extremely large numbers are used). For example:

```
a = 2**100 #=> 1267650600228229401496703205376
```

Python is not susceptible to the vulnerability associated with shifting the underlying number as described in ISO/IEC 24772-1:2024 6.3 because Python 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
```

There is no overflow check required for left shifts since bits are added as required. For right shifts of positive numbers, the result will decrease by powers of two with a limit

of zero. Note that right shifts of negative numbers eventually result in -1 if the number of positions shifted is sufficiently large.

The vulnerability associated with endianness can be mitigated by identifying the endian protocol. Use `sys.byteorder` to determine the native byte order of the platform. The call returns `big` or `little`.

6.3.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Follow the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.3.5.
- Be careful when shifting negative numbers to the right as the number will never reach zero.
- Localize and document the code associated with explicit manipulation of bits and bit fields.
- Use `sys.byteorder` to determine the native byte order of the platform.

6.4 Floating-point arithmetic [PLF]

6.4.1 Applicability to language

The vulnerabilities described in ISO/IEC 24772-1:2024 6.4 apply to Python.

Python supports floating-point arithmetic with a specified mantissa of 53 bits. Literals are expressed with a decimal point and or an optional `e` or `E`:

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

Python provides decimal fixed-point and floating-point libraries for use where appropriate.

6.4.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Follow the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.4.5.
- Code algorithms to account for the fact that results can vary slightly by implementation.

6.5 Enumerator issues [CCB]

6.5.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.5 partially apply to Python.

An `enum` module was introduced in Python v3.4 which allows for better iteration and value comparison than most previous user-developed methods. An example of the new `enum` module is:

```
from enum import Enum
class ColorEnum(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
    YELLOW = 4
print(ColorEnum.BLUE) #=> ColorEnum.BLUE

from enum import Enum
class ColorEnum(Enum):
    RED = 1
    GREEN = 3
    BLUE = 2
    YELLOW = 4
print(ColorEnum.BLUE)
print(ColorEnum.GREEN.value > ColorEnum.BLUE.value) #=> TRUE
```

Values can be assigned to the names either manually or automatically using `auto()`. Using `auto()` ensures that each name is assigned a unique and sequential value and the initial assignment starting at 1 (not 0).

```
from enum import Enum, auto
class ColorEnum(Enum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()
    YELLOW = auto()

for color in ColorEnum:
    print(color.value) #=> 1,2,3,4
```

If values are assigned manually, they can occur out of sequence, but care must be taken to ensure that there are no repeat values since only the first unique value is recognized and all subsequent repeated values are ignored. For example:

```
from enum import Enum
class ColorEnum(Enum):
    RED = 1
    GREEN = 2
    BLUE = 2
    YELLOW = 3
for color in ColorEnum:
    print(color.name, color.value) #=> RED 1, GREEN 2,
                                   _ #=> YELLOW 3
```

Deleted: ¶

Notice that `BLUE` is completely ignored since it is a repeated value. Duplicate values can be detected and forced to raise a `ValueError` by using the `@unique` class decorator as shown below:

```
from enum import Enum, unique
@unique
class ColorEnum(Enum):
    RED = 1
    GREEN = 2
    BLUE = 2
    YELLOW = 3

for color in ColorEnum:
    print(color.name, color.value)
           #=> ValueError:duplicate values found in
           #=> <enum 'ColorEnum'>: BLUE -> GREEN
```

Mixing `auto()` with manual assignments can be prone to error for the same reason. For example:

```
from enum import Enum, auto
class Colors(Enum):
    RED = auto()
    BLUE = auto()
    GREEN = auto()
    PURPLE = 0
    YELLOW = 1
print(list(Colors)) #=> [<Colors.RED:1>, <Colors.BLUE:2>,
                    #=> <Colors.GREEN:3>, <Colors.PURPLE:0>]
```

Notice that `YELLOW` is missing since its manually assigned value of 1 had already been created automatically.

Another interesting scenario that involves lists and `auto()` is shown here:

```
from enum import IntEnum, auto
colors = ["RED", "GREEN"]
class Nums(IntEnum):
    ONE = auto()
    TWO = auto()
    THREE = auto()
print(colors[Nums.ONE]) #=> GREEN
```

On the other hand,

```
print(colors[Nums.ONE-1]) #=> RED
```

Notice that in this scenario the first item in the colors list (`RED`) cannot be accessed using `auto()`, unless `1` is subtracted from every enumeration constant created by `auto()`.

Given that enumeration is a useful programming device, many programmers choose to implement their own enumeration objects or types using a wide variety of methods including the creation of classes, lists, and even dictionaries. Such substitutes carry the risk that the usual enumeration semantics will be incompletely implemented.

In Python releases before 3.4, programmers used various other Python capabilities to implement the functionality of enumerations, each with its own set of vulnerabilities. New programs should use the provided functionality of `enum` as it is a more complete implementation. Programs created before Python 3.4 can consider updating their relevant code to use the `enum` module. For example, sets of strings can be used to simulate enumerations:

```
colors = ['red', 'green', 'blue']
if ‘red’ in colors:
    print('Valid color') #=> Valid color
```

6.5.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Follow the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.5.5.
- Use type annotations to help provide static type checking prior to running the code.
- Avoid the use of `auto()` for enums intended to be used for indexing into lists.
- If using `auto()` for defining enums, ensure that `auto()` is used everywhere.
- If using `auto()` for defining enums, be very careful in converting to list members.

6.6 Conversion errors [FLC]

6.6.1 Applicability to language

The vulnerabilities identified in ISO/IEC 24772-1:2024 6.6 apply to Python, except those related to integer-based conversions since Python seamlessly handles integers as described below.

Python has updated how it handles coercion and instead of using the “lifting” technique that brings operands to a common type, it leaves the handling of different operand types to the operation. If a style slot is incapable of handling an argument type combination, the `Py_NotImplemented` singleton signals to the caller that the operation is not implemented for the type combination. This signals the caller to try other operation slots until it finds one that is compatible with the type combination being implemented. If there are no compatible combinations found, a `TypeError` exception is raised.

Native Python numerical types are converted using the following rules:

- If either argument is a complex number, the other is converted to the complex type;
- Otherwise, if either argument is a floating-point number, the other is converted to floating-point;
- Otherwise, both must be plain integers and no conversion is necessary.

Integers in the Python language are of a length bounded only by the amount of memory in the machine. Implementations may store integers in an internal format that has faster performance when the number is smaller than the largest integer supported by the implementation language and platform, but this detail is not exposed to the language user in Python.

Converting from a floating-point number to an integer, either implicitly (using the `int()` function) or explicitly, will typically cause a loss of precision:

```
a = 3.0
print(int(a)) #=> 3 (no loss of precision)
a = 3.1415
print(int(a)) #=> 3 (precision lost)
```

Precision can also be lost when converting from a very large integer to a floating-point number where it requires more than 53 bits of precision. Losses in precision, whether from an integer to floating-point conversion or vice versa, do not generate errors but can lead to unexpected results especially when floating-point numbers are used for loop control.

Conversions of an excessively large integer or their string equivalent to a float will lead to the exception `OverflowError` (see 6.36 Ignored error status and unhandled exceptions [OYB]).

Explicit conversion methods can also be used to convert between types though this is seldom required for numbers since Python will automatically convert as required. Examples include:

```
a = int(1.6666) #=> 1
b = float(1)    #=> 1.0
c = int('10')   #=> 10
d = str(10)     #=> '10'
e = ord('x')    #=> 120
f = chr(121)    #=> 'y'
```

Conversions between unrelated types are not possible in Python. For conversions up and down a class hierarchy, see 6.44 Polymorphic variables [BKK].

6.6.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Follow the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.6.5.

- Though there is generally no need to be concerned with an integer getting too large (rollover) or small, be aware that iterating or performing arithmetic with very large positive or small (negative) integers will hurt performance.
- Be aware of the potential consequences of precision loss when converting from floating-point to integer.
- Design coding strategies that allow the distinction of semantically incompatible types.
- Design classes that have operation handling methods carefully and ensure that `Py_NotImplemented` and `TypeError` exceptions are handled.
- Use or develop `units` libraries to handle conversions between differing unit-based systems.

6.7 String termination [CJM]

6.7.1 Applicability to language

This vulnerability is not applicable to Python native programming, as Python does not use null terminated strings. Python strings are immutable objects whose length can be queried with built-in functions. Therefore, Python raises an exception for any access past the end or beginning of a string.

```
a = '12345'
b = a[5] #=> IndexError: string index out of range
```

Vulnerabilities associated with runtime exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

Python programs, however, may include extension modules written in C or C++, and any string types used for those modules will be C-based string types which have the vulnerability.

6.7.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.7.5.

- Where C style strings or C++ style strings are used, apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 .

6.8 Buffer boundary violation [HCB]

This vulnerability is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary. Vulnerabilities associated with runtime exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

6.9 Unchecked array indexing [XYZ]

The vulnerability as described in ISO/IEC 24772-1:2024 6.9 is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary. Vulnerabilities associated with runtime exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

6.10 Unchecked array copying [XYW]

The vulnerability as described in ISO/IEC 24772-1:2024 6.10 is not applicable to Python because assigning lists is done by reference. A deep copy of a list creates a new list object. There is a potential vulnerability associated with copying an object over part of itself when an object is complex, such as lists of lists (see 6.38 Deep vs. shallow copying [YAN]).

6.11 Pointer type conversions [HFC]

6.11.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.11 are applicable to Python since, although Python does not have traditional visible references to memory (pointers), every variable contains an implicit pointer to the actual object. Additionally, Python permits code to instruct instances to misrepresent their type. Consuming code always has the option to decide whether to believe the real type or the claimed type, but naive code will believe any claims by default. The following example illustrates how an object's type can be misrepresented during runtime:

Commented [SJM5]: It might be useful to point out the following:
- Python doesn't have *explicit* (traditional) pointers like many other languages, but rather does have implicit pointers
- Every variable in Python is a pointer, because variables in Python are names that refer to objects

Commented [SM6R5]: Implemented.

Commented [SJM7]: Lie="to make an untrue statement with *intent* to deceive." This may be a little strong, possibly modify:

…misrepresent their type.

Commented [SJM8R7]:

```
class Example:
    def method(self):
        print("From Example: ", type(self), self.__class__)

class Other:
    def method(self):
        print("From Other: ", type(self), self.__class__)

x = Example()
x.method()           #=> From Example:  <class '__main__.Example'>
x.__class__ = Other  #=> Reassign the type of the current x instance
x.method()           #=> From Other:  <class '__main__.Other'>
```

6.11.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.11.5.
- Forbid altering the `__class__` attribute for instances of a class unless there are compelling reasons to do so. If alterations are required, document the reasons in docstring and local comments.
- Use type annotations and type hints in the code.
- Run a third-party static type-checker.

6.12 Pointer arithmetic [RVG]

This vulnerability as documented in ISO/IEC 24772-1:2024 6.12 is not applicable to Python because Python does not have pointers and does not permit arithmetic on references.

6.13 Null pointer dereference [XYH]

This vulnerability as documented in ISO/IEC 24772-1:2024 6.13 does not apply to Python. The Python equivalent of a null pointer is the object `None`. Accessing this

object raises an exception. Vulnerabilities associated with runtime exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

6.14 Dangling reference to heap [XYK]

6.14.1 Applicability to language

These vulnerabilities as documented in ISO/IEC 24772-1:2024 6.14 only minimally apply to Python because Python exclusively uses garbage collection for memory reclamation, thus no dangling references can exist. Specifically, Python only uses namespaces to access objects, therefore when an object is deallocated there are no names denoting the reclaimed object. Attempts to access those names anyway will raise runtime exceptions as usual. Vulnerabilities associated with runtime exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

Note that due to reference cycles and `__del__` methods, it is possible for objects that were scheduled for deallocation to gain new live references, and hence not be candidates for deallocation after all. Python runtimes are aware of this when it happens, and avoid deallocating the memory, ensuring that dangling references to heap memory are not created.

Python permits direct access to the internal data of objects by using the `memoryview()` function. The `memoryview()` function is useful on very large objects since it does not create a copy of the object data and, as a result, can perform certain tasks much faster. Managing this direct access to objects does require verification that the object data remains valid even if the object is no longer needed elsewhere in the program.

6.14.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.14.5.
- When accessing data objects directly by using `memoryview()`, make sure that the data pointed to remains valid until it is no longer needed.

6.15 Arithmetic wrap-around error [FIF]

6.15.1 Applicability to language

The vulnerabilities discussed in ISO/IEC 24772-1:2024 6.15.3 do not apply to Python for integers.

Operations on integers in Python cannot cause wrap-around errors because integers have no maximum size other than what the memory resources of the system can accommodate.

Shift operations operate correctly, except that large shifts on negative numbers infill with '1's and will often result in a final answer of `-1`.

Normally the `OverflowError` exception is raised for floating-point wrap-around errors but, for implementations of Python written in C, exception handling for floating-point operations cannot be assumed to catch this type of error because they are not standardized in the underlying C language. Because of this, most floating-point operations cannot be depended on to raise this exception.

Attempts to convert large integers that cannot be represented as a double-precision ISO/IEC 60559 value to float will raise `OverflowError`.

```
bigint = 2 * 10 ** 308
float(bigint) #=> OverflowError: int too large to convert to
float
```

The vulnerabilities associated with unhandled exceptions are discussed in 6.36 Ignored error status and unhandled exceptions [OYB].

6.15.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.15.5.
- Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap-around errors.
- Avoid using floating-point or decimal variables for program flow logic, but if one of these types must be used, then bound loop structures to not exceed the maximum or minimum possible values for the loop control variables.
- Test the implementation that is being used to see if exceptions are raised for floating-point operations and if they are then used for exception handling to catch and handle wrap-around errors.

Commented [SJM9]: Reword
And if they are then used for exception handling...

6.16 Using shift operations for multiplication and division [PIK]

This vulnerability is not applicable to Python because there is no practical way to overflow an integer since integers have unlimited precision, left shifts are defined in terms of multiplication by powers of 2, and right shifts are defined in terms of floor division by powers of two.

```
print(-1 << 100) #=> -1267650600228229401496703205376
print(1 << 100)  #=> 1267650600228229401496703205376
print(-4 >> 3)   #=> -1 where 0 might be expected
```

6.17 Choice of clear names [NAI]

6.17.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.17 exist in Python.

Python provides very liberal naming rules:

- Names may be of any length and consist of letters, numerals, and underscores only. All characters in a name are significant. Note that unlike some other languages where only the first n number of characters in a name are significant, all characters in a Python 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 names must start with an underscore or a letter.
- Names are case sensitive, for example, 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, for example, aLpha versus alpha.
- Names allow all Unicode “script” code points to be used as letters, and each numerical code point is considered distinct when used as part of a name, even if their visual rendering is similar. Some Unicode characters can cause confusion for humans in that what they read may not be the text that is processed by the language processor. For example, using homoglyphs, Confused (Cyrillic ES) versus Confused (Latin C), or aIpha (Latin capital I) versus alpha (Latin lowercase l) will be different names.

The following naming conventions are not part of the standard but are in common use:

- Class names start with an upper-case letter, all other variables, functions, and modules are in all lower case.
- Names starting with a single underscore (_) are not imported by the "from *module* import *" statement – this not part of the standard but most implementations enforce it.
- Names starting and ending with two underscores (__) are system-defined names.
- Names starting with, but not ending with, two underscores are local to their class definition.
- Python provides a variety of ways to package names into namespaces so that name clashes can be avoided:
    - Names are scoped to functions, classes, and modules meaning there is normally no collision with names utilized in outer scopes and vice versa.
    - Names in modules (a file containing one or more Python statements) are local to the module and are referenced using qualification (for example, a function `x` in module `y` is referenced as `y.x`). Though local to the module, a module's names can be, and routinely are, copied into another namespace with a "`from module`" import statement.

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

- Names are not required to be 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 raise an exception (see 6.22 Initialization of variables [LAV]).
- Names can be unique but may look similar to other names, for example, `alpha` and `aLpha`, `__x` and `_x`, `_beta__` and `__beta_` which could lead to the use of the wrong variable. Python will not detect this problem as it parses the expression.

Python utilizes dynamic typing with types determined at runtime. There are no type or variable declarations for an object by default, which can lead to subtle and potentially catastrophic errors:

```
x = 1
# lots of code…

# and eventually

X = 10
```

In the code above, the programmer intended to set (lower case) `x` to 10 and instead created a new (upper case) `X` with the value `10` and leave lower-case `x` unchanged. Python will not detect a problem because it is a case-sensitive language and every change of case in a name will result in a new object. For example, `THIS`, `This`, `THis`, and `this` are all different variables.

6.17.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.17.5.
- For more guidance on Python's naming conventions, refer to Python Style Guides contained in "PEP 8 – Style Guide for Python Code"[10].
- Avoid names that differ only by case unless necessary to the logic of the usage, and in such cases document the usage.
- Adhere to Python's naming conventions.
- Avoid 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 because the compiler cannot assist in detecting names that appear similar but are different.
- Ensure that 'show-all-hidden-characters' is enabled in the editor.
- Understand or eliminate all confusing Unicode characters, in particular, homoglyphs.
- Use caution when copying and pasting Unicode text.

6.18 Dead store [WXQ]

6.18.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.18 applies to Python, since it is possible to assign a value to a variable and never reference that variable which causes a "dead store". This in itself is not harmful, other than the memory that it wastes,

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

Similarly, if dead stores cause the retention of critical resources, such as file descriptors or system locks, then this retention may cause subsequent system failures.

Variables local to a function are deleted automatically when the encompassing function is exited but, though not a common practice, variables can be explicitly deleted when they are no longer needed using the `del` statement.

6.18.2 Avoidance mechanisms for users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.18.5.
- Assume that when examining code, that a variable can be bound (or rebound) to another object (of same or different type) at any time.
- Avoid rebinding except where it adds identifiable benefit.
- Consider using `ResourceWarning` to detect implicit reclamation of resources.

6.19 Unused variable [YZS]

6.19.1 Applicability to language

The vulnerability as described in ISO IEC TR 24772-1 6.19 is applicable to Python.

6.19.2 Avoidance mechanisms for language users

Software developers can avoid the vulnerability or mitigate its ill effects by applying the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.19.5.

6.20 Identifier name reuse [YOW]

6.20.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1 6.20 apply to Python.

Python has the concept of namespaces which are simply the places where names exist in memory. Namespaces are associated with functions, classes, and modules. When a name is created (that is, when it is first assigned a value), it is associated (that is,

bound) to the namespace associated with the location where the assignment statement is made (for example, in a function definition). The association of a variable to a specific namespace is elemental to how scoping is defined in Python.

Scoping allows for the definition of more than one variable with the same name to reference different objects. For example:

```
avar = 1
def x():
    avar = 2
    print(avar) #=> 2
x()
print(avar) #=> 1
```

The variable `avar` 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 `avar` then it would need to specify that `avar` was a global before referencing it as in:

```
avar = 1
def x():
    global avar
    avar = 2
    print(avar) #=> 2
x()
print(avar) #=> 2
```

In the case above, the function is updating the variable `avar` that is defined in the calling module. There is a subtle but important distinction on the locality versus global nature of variables: assignment is always local unless global is specified for the variable as in the example above where `avar` is assigned a value of `2`. If the function had instead simply referenced `avar` without assigning it a value, then it would reference the topmost variable `avar` which, by definition, is always a `global`:

```
avar = 1
def x():
    print(avar)
x() #=> 1
```

The rule illustrated above is that attributes of modules (that is, variable, function, and class names) are global to the module meaning any function or class can reference them.

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

- A nested function's variables are in the scope of the nested function only.
- Variables defined in a module are in global scope, which means they are scoped to the module only and are therefore not visible within functions defined in that module (or any other function) unless explicitly identified as `global` at the start of the function.

Python has ways to bypass implicit scope rules:

- The `global` statement, which allows an inner reference to an outer scoped variable(s).
- The nonlocal statement, which allows a variable in an enclosing function definition to be referenced from a nested 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 Python, one must be explicit when intending to circumvent the intrinsic scoping of variable names. The downside is that identical variable names, which are totally unrelated, can appear in the same module, which could lead to confusion and misuse unless scoping rules are well understood.

Names can also be qualified to prevent confusion as to which variable is being referenced:

```
avar = 1
class xyz():
    avar = 2
    print(avar)        #=> 2

print(xyz.avar, avar) #=> 2 1
```

The final print function call above references the `avar` variable within the `xyz` class and the global `avar`.

6.20.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.20.5.

- Forbid the use of 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 annex and because their bypassing of standard scoping rules make the code harder to understand.
- Use qualification when necessary to ensure that the correct variable is referenced.

6.21 Namespace issues [BJL]

6.21.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.21 are applicable to Python when modules are imported.

Python has a hierarchy of namespaces, which provides isolation to protect from name collisions, ways to explicitly reference down into a nested namespace, and a way to reference up to an encompassing namespace. Generally speaking, namespaces are isolated. For example, a program's variables are maintained in a separate namespace from any of the functions or classes it defines or uses. The variables of modules, classes, or functions are also maintained in their own protected namespaces. Namespaces may be nested.

For certain scenarios, the local namespace is dictated by the order of importation. For example, the scenarios below import two files (`a.py` and `b.py`) and each file contains a function named `meth()`. Importing the files using `from x import *` results in the last import to be used. In the second scenario, using only the `import x` method allows the use of either `meth()` by prefacing it with the desired library name regardless of order presented in the file.

```
< -  file = a.py - >
def meth():
    print("From A")

< -  file = b.py - >
def meth():
    print("From B")
------------------------
 from a import *
 from b import *
 from a import *
 meth() #=> From A
 -------------------------
 import a
```

```
import b
a.meth() #=> From A
```

See 6.41 Inheritance [RIP] for a discussion of multiple inherited methods with the same name.

Accessing a namespace's attribute (that is, a variable, function, or class name), is generally done in an explicit manner to make it clear to the reader (and Python) which attribute is being accessed:

```
n = Animal.num # fetches a class' variable called num
x = mymodule.y # fetches a module's variable called y
```

The examples above exhibit qualification – there is no doubt from where a variable is being fetched. Qualification can also occur from an encompassed namespace up to the encompassing namespace using the `global` statement:

```
def x():
    global y
    y = 1
```

The example above uses an explicit `global` statement which makes it clear that the variable `y` is not local to the function `x`; it assigns the value of `1` to the variable `y` in the encompassing module.

Python also has some subtle namespace issues that can cause unexpected results especially when using imports of modules. For example, assuming module `M1.py` contains:

```
a = 1
```

And module `M2.py` contains:

```
b = 1
```

Executing the following code is not a problem since there is no variable name collision in the two modules (the "`from <modulename> import *`" statement brings all attributes of the named module into the local namespace):

```
from M1 import *
print(a) #=> 1
```

```
from M2 import *
print(b) #=> 1
```

Later, the author of the `M2` module adds a variable named `a` and assigns it a value of `2`. Now `M2.py` contains:

```
b = 1
a = 2 # new assignment
```

The programmer of module `M2.py` can lack knowledge of the module `M1.py` or the information that the program imports both `M1` and `M2`. The importing program, with no changes, is run again:

```
from M1 import *
print(a) #=> 1
from M2 import *
print(a) #=> 2
```

The results are now different because the importing program is susceptible to unintended consequences due to changes in variable assignments made in two unrelated modules as well as the sequence in which they were imported. Also note that the "`from <modulename> import *`" statement brings all of the module's attributes into the importing code which can silently overlay like-named variables, functions, and classes.

A common surprise of the Python language is that Python detects local names (a local name is a name that lives within a class or function's namespace) statically by looking for one or more assignments to a name within the class/function. If one or more assignments are found, then the name is noted as being local to that class/function. This can be confusing because if only references to a name are found then the name is referencing a global object so the only way to know if a reference is local or global, barring an explicit global statement, is to examine the entire function definition looking for an assignment. This runs counter to Python's goal of "Explicit is better than implicit" (EIBTI):

```
a = 1
def f():
     print(a)   #a is local
     a = 2
f() #=> UnboundLocalError: local variable 'a' referenced
before
    #   assignment

# now with the assignment commented out
a = 1
```

```
def f():
      print(a) #=> 1    #a is global
      #a = 2

# Assuming a new session:
a = 1
def f():
    global a
    a = 2 * a
f()
print(a) #=> 2
```

Note that the rules for determining the locality of a name applies to the assignment operator "=" as above, but also to all other kinds of assignments which includes module names in an import statement, function and class names, and the arguments declared for them (see 6.19 Unused variable [YZS]).

Python can perform either absolute or relative imports. An absolute import specifies the resource to be imported using its full path from the project's root folder. A relative import specifies the resource is to be imported relative to the current location. Although the full path of an import can be long, the use of an absolute import defines explicitly what resource is being imported.

Name resolution follows a simple Local, Enclosing, Global, Built-ins (LEGB) sequence:

- First the local namespace is searched;
- Then the enclosing namespace (that is, a `def` or a `lambda` expression), recursively;
- Then the global namespace;
- Lastly the built-in namespace.

Python v3.3 introduced `types.prepare_class()` which gives more control over how classes and metaclasses are created. The `__prepare__` function can be called prior to the creation of a metaclass instance giving complete control over how the class declarations are ordered. It also allows symbols to be inserted into the class namespace, which can be used elsewhere in the class, but these inserted symbols are only visible during class construction.

6.21.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

Commented [p10]: These what?

Commented [SJM11R10]: As a refresher, this content was derived from PEP 3115 https://peps.python.org/pep-3115/

"these" is simply referring to the mentioned symbols, but this could probably be worded more clearly (good point).

Metaclasses are an advanced area of Python but can be useful in certain circumstances. For example, metaclasses can be used to create function overloading in Python since, by default, Python does not inherently have this capability. *However*, we may want to reconsider including this paragraph since we offer no concrete guidance when using metaclasses in 6.21.2 other than to state the use of `__prepare__` (which merely opens the door to this capability).

There are many other advanced capabilities that we have deemed to be beyond the scope of this document, and the safe use of metaclasses may also fall into this category. We can discuss this further. If we do decide to keep this paragraph, and potentially add an example, it may end up being tutorial in nature.

If interested, here are some useful videos on the topic:

https://www.youtube.com/watch?v=NAQEj-c2CI8

https://www.youtube.com/watch?v=yWzMiaqnpkI

- Use the full path name for imports, in preference to relative paths.
- When using the import statement, rather than use the `from x import *` form (which imports all of module `x`'s attributes into the importing program's namespace), instead explicitly name the attributes that need to be imported (for example, `from X import a, b, c`) so that variables, functions and classes are not inadvertently overlaid.
- Avoid implicit references to global values from within functions to make code clearer. In order to update globals within a function or class, place the global statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are global (for example, `global a, b, c`).
- When interfacing with external systems or other objects where the declaration order of class members is relevant, use `__prepare__` to obtain the desired order for class member creation.

6.22 Missing initialization of variables [LAV]

6.22.1 Applicability of language

This vulnerability applies only minimally to Python because all attempts to access an uninitialized variable result in an exception. There is no ability to use a variable with an uninitialized value because assigned variables always reference objects which always have a value and unassigned variables do not exist. Therefore, Python raises an exception at runtime when a name that is not bound to an object is referenced.

Static type analysis tools can be used prior to execution to identify accesses to names that are not bound to objects.

Vulnerabilities associated with runtime exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

6.22.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.22.5.
- Ensure that it is not logically possible to reach a reference to a variable before it is assigned to avoid the occurrence of a runtime error.

6.23 Operator precedence and associativity [JCW]

6.23.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.23 applies to Python.

Python provides many operators and levels of precedence, so it is not unexpected that operator precedence and associativity are not well understood and hence misused. For example:

```
2 ** 2 ** 3       # Yields 256, not 64 (right-
                  # associativity)

c and a==b        # parses as c and (a==b)
```

Avoidance mechanisms for language users

Software developers can avoid the vulnerability or mitigate its ill effects by applying the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.23.5.

6.24 Side-effects and order of evaluation of operands [SAM]

6.24.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.24 exists in part in Python. Operands are evaluated left-to-right in Python and hence the evaluation order is deterministic, but the vulnerabilities associated with short-circuit operators exist in Python. Additional vulnerabilities arise from Python semantics of loops that alter data structures.

Some of Python's data structures such as `list`, `dict`, `set`, and `bytearray` are mutable. Attempting to delete items from one of these data structures, from within a loop, are liable to result in undesirable side-effects. The example below shows that using the loop index to delete items in the numbers list produces unexpected results since the loop index `i` is based on the full length of the original list but gets modified within the loop.

Commented [SJM12]: may

Commented [SJM13]: 'produces unexpected results' … it is not incorrect but rather an unexpected result due to Python's handling of this situation

```
nums = [1, 2, 2, 3, 4, 5]
for i in nums:
    if i & 1 == 0: # remove even numbers
        nums.remove(i)
print(nums) # => [1, 2, 3, 5]
```

The above output is unexpected, as it also contains even numbers. The correct approach is to create a copy of the original list by using the [:] operator as shown below:

```
nums = [1, 2, 2, 3, 4, 5]
for i in nums[:]:
    if i & 1 == 0: # remove even numbers
        nums.remove(i)
print(nums) # => [1, 3, 5]
```

Numeric data types in Python are immutable and remain unchanged when used as an argument within a calling function. However, if the immutable argument within a calling function is made to be a global variable, then that argument is changed even though it is usually an immutable type. This potentially unexpected side-effect is illustrated in the following example. The `double` function call passes the immutable integer "`y`" as an argument to the `double` function, but because it is declared as a `global` variable within the function, the integer argument that is typically immutable is modified.

```
def double(n):
   global y
   y = 2 * n

y = 5
double(y)
print(y) #=> 10
```

Potentially unexpected side-effects can also be experienced by changing an external list in a loop. For example, the following code shows that adding the color `black` to the `colors` list updates the list since lists are mutable objects. The `for` loop recognizes this new list member and continues with another pass through the loop with the index counter `i` now set to black resulting in the color `white` being added to the `colors` list.

Commented [SJM14]: The Odd numbers are not unexpected. See

Formatted: Font: (Default) Cambria, 12 pt, English (CAN)

Formatted: Font: (Default) Cambria, 12 pt, English (CAN)

Formatted: Font: (Default) Cambria, 12 pt, English (CAN)

Commented [SJM15]: Reword?

The normally ummutable argument…

The argumrnt that is typically immutable is modified ….

Commented [SM16R15]: OK

```
colors = ["red"]
for i in colors:
    if i == "red":
        colors += ["black"]
    if i == "black":
        colors += ["white"]
print(colors) #=> ['red', 'black', 'white']
```

To avoid the unexpected side effects, is it recommended to use a copy of the list within the loop. In this scenario, black is added to the local `colors` list but since the loop index `i` never takes on a value other than `red`, the color `white` is never added to the `colors` list.

```
colors = ["red"]
for i in colors[:]: # Avoid side effects by using a local
list
    if i == "red":
        colors += ["black"]
    if i == "black":
        colors += ["white"]
print(colors) #=> ['red', 'black']
```

Python allows reassignment of loop indexes, which can lead to unexpected results depending on the order of reassignment. For example, the following code illustrates two scenarios where the loop index "`i`" is reassigned within a loop. The first scenario uses the loop index prior to reassignment and prints out the expected sequence. The second scenario uses the loop index after reassignment and, since it creates a new object with a value of 10, this new value is printed out. Internally, the loop index counter remains intact, and the loop exits after four iterations as expected.

```
for i in range(1, 5):
    print(i) #=> 1,2,3,4
    i = 10

for i in range(1, 5):
    i = 10 # new i is created, doesn't affect the loop count
    print(i) #=> 10,10,10,10
```

Python supports sequence unpacking (parallel assignment) in which each element of the right-hand side (expressed as a tuple) is evaluated and then assigned to each

element of the left-hand side (LHS) in left-to-right sequence. For example, the following is a safe way to exchange values in Python:

```
a = 1
b = 2
a, b = b, a # swap values between a and b
print (a, b) #=> 2, 1
```

Assignment of the targets (LHS) proceeds left-to-right so overlaps on the left side are not safe:

```
a = [0,0]
i = 0
i, a[i] = 1, 2 #=> Index is set to 1; list is updated at [1]
print(a) #=> 0,2
```

It is possible to have unintended results when the variables on the left side overlap with one another. Therefore, it is important to ensure that the assignments and left-to-right sequence of assignments to the variables on the left-hand side do not overlap. To avoid this situation, consider breaking the statement into two or more statements:

```
# Overlapping
a = [0,0]
i = 0
i, a[i] = 1, 2   # Index is set to 1; list is updated at [1]
print(a)         #=> 0,2

# Non-overlapping
a = [0,0]
i, a[0] = 1, 2
print(a)         #=> 2,0
```

Python Boolean operators are often used to assign values as in:

```
a = b or c or d or None
```

Variable `a` is assigned the first value of the first object that has a non-zero (that is, `True`) value or, in the example above, the value `None` if `b`, `c`, and `d` are all `False`. This is a common and well understood practice. Difficulty can arise, however, if a value such as 5 (for c) is included, then `a` will receive the value 5 instead of True or False.

As with many languages, Python performs short-circuiting in Boolean expressions. In the case of "`x or y`", Python only evaluates `y` if `x` evaluates to `False`. Likewise, for "`x`

Commented [SJM17]: This is probably OK as is, but we could elaborate on this example.

```
b=d=0
c=5
a = b or c or d or None
print(a) #=> 5
```

INTERESTING: The operands of an expression involving a boolean expression (OR, AND, etc.) would expectedly have Boolean values, but objects in Python are not very strict about this and internally implements a set of rules to decide if an object is considered true or false

https://docs.python.org/3/library/stdtypes.html

"By default, an object is considered true unless its class defines either a __bool__() method that returns False or a __len__() method that returns zero, when called with the object. [1] Here are most of the built-in objects considered false:

- constants defined to be false: None and False
- zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- empty sequences and collections: '', (), [], {}, set(), range(0)"

Commented [SM18R17]: OK. SM

Commented [p19]: Belongs further up where there is an example of short-circuiting already.

Commented [SJM20R19]: Concur

Commented [SM21R19]: OK. Maybe here?

`and y`", Python only evaluates `y` if `x` is `True`. Trouble can be introduced when functions or other constructs with side effects are used on the right side of a Boolean operator:

```
if a() or b()
```

If function `a` returns a `True` result then function `b` will not be called which may cause unexpected results if function `b` has side effects. If necessary, perform each expression first and then evaluate the results:

```
x = a()
y = b()
if x or y …
```

6.24.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.24.5.
- Avoid assignment to a variable equally named as a loop index counter within the loop.
- Be aware of Python's short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression.
- Avoid any operation that changes the size of a data structures while iterating over it and instead, create a new list.

6.25 Likely incorrect expression [KOA]

6.25.1 Applicability to language

The vulnerabilities as described in TR 24772-1 6.25 apply to Python, but Python goes to some lengths to help prevent some of the likely incorrect expressions.

Testing for equivalence cannot be confused with assignment and improper use will result in error, for example:

```
a = b = 1
if (a=b): print(a, b) #=> SyntaxError: invalid syntax.
                      #=> Maybe you meant '==' or ':='
if (a==b): print(a, b) #=> 1 1
```

Boolean operators use English words `and`, `not`, `or`. Bitwise operators use symbols `&`, ~, and `|`, respectively.

Python, however, does have some subtleties that can cause unexpected results:

— Skipping the parentheses after a function does not invoke a call to the function and can fail silently because it is a legitimate reference to the function object:

```
class a:
      def demo():
            print("in demo")

a.demo() #=> in demo
a.demo   # <function demo at 0x000000000342A9C8>
x = a.demo
x()      #=> in demo
```

The two lines that reference the function without trailing parentheses above demonstrate how that syntax is a reference to the function object and not a call to the function.

— Built-in functions that perform in-place operations on mutable objects (that is, lists, dictionaries, and some class instances) do not return the changed object – they return `None`:

```
a = []
a.append("x")
print(a) #=> ['x']
a = a.append("y")
print(a) #=> None
```

— In `async` code, forgetting to use an `await` statement results in a warning about the unawaited coroutine.

Short-circuit operations can be a source of likely incorrect expressions as described in 6.24 “Side effects and order of evaluation of operands [SAM]”.

Commented [p22]: Interesting! What about a = b == 1 as the intended code?

Commented [SJM23R22]: a = b == 1
^
NameError: name 'b' is not defined

Commented [SJM24]: More Completely:

SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?

Commented [SM25R24]: Done

6.25.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.25.5.
- Add parentheses after a function call in order to invoke the function.
- Keep in mind that any function that changes a mutable object in place returns a `None` object – not the changed object since there is no need to return an object because the object has been changed by the function.
- Be aware of the difference between equality (`==`) and identity (`is`) and use them as appropriate.

6.26 Dead and deactivated code [XYQ]

6.26.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.26 applies to Python.

There are many ways to have dead or deactivated code occur in a program and Python is no different in that regard. Except in very limited cases, Python does not provide static analysis to detect such code nor does the very dynamic design of Python's language lend itself to such analysis. The limited cases are those where a known-false constant value (for example `0`, `False`) is used directly in a conditional flow control check (the branch will never be taken, so code does not need to be emitted for it), and when a function unconditionally executes a return statement (no code needs to be emitted for the section after the function returns).

Python supports type hints (see 5.1.3) that can be used along with third party static analysis tools to detect dead or deactivated code.

The module and related `import` statement provide convenient ways to group attributes (for example, functions, names, and classes) into a file which can then be copied, in whole, or in part (using the `from` statement), into another Python module. All of the attributes of a module are copied when either of the following forms of the `import` statement is used. This is roughly equivalent to simply copying in all of code directly into the importing program, which can result in code that is never invoked (for example, functions which are never called and hence "dead"):

```
import modulename
```

Commented [p26]: True, but equally true for non-top-level returns for the code up to the next join.

Commented [SJM27R26]: This entire paragraph warrants reconsideration in my opinion.

Per the text:
"*Except in very limited cases, Python does not provide static analysis* to detect such code …".

Strictly speaking, static analysis is not a capability of a language, but rather the result of compilers, linkers and 3rd-party tools such as pyflakes and vulture (for Python when hints are used). I do agree with suggesting the use of Python-specific static analysis tools for finding dead code, but we may want to reword this content so that it does not come across as a feature that Python, by itself, has.

Type hints are discussed in Section 5.1.3 and recommended in numerous other locations.

Citing an example of dead code in this paragraph is fine, but not necessary in my opinion (tutorial).

The following example runs successfully without warning:

```
if False:
    print('hello from False')
if True:
    print('hello from True'
```

OUTPUT:
```
hello from True
```

Commented [SM28R26]: Done.

```
from modulename import *
```

The `import` statement in Python loads a module into memory, compiles it into byte code, and then executes it. Subsequent executions of an `import` for that same module are ignored by Python and have no effect on the program whatsoever. The reload statement is required to force a module, and its attributes, to be loaded, compiled, and executed.

6.26.2Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.26.5.
- Import just the attributes that are required by using the `from` statement to avoid adding dead code.
- Be aware that subsequent imports of the same module have no effect; use the `reload` statement instead of `import` if a fresh copy of the module is desired.

6.27 Switch statements and static analysis [CLL]

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.27 do not apply to Python, which does not have a switch statement nor the concept of labels or branching to a demarcated “place”.

6.28 Demarcation of control flow [EOJ]

6.28.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.28 only minimally apply to Python. Python makes demarcation of control flow very clear because it uses indentation (using spaces or tabs – but not both within a given code block) as the only demarcation construct:

```
a, b = 1, 1
if a:
    print("a is True")
else:
    print("False")
    if b:
        print("b is true")
print("back to main level")
```

The code above prints “`a is True`” followed by “`back to main level`”. Note how control is passed from the first `if` statement’s True path to the main level based

entirely on indentation while in other languages that do not rely on indention, the second `if` statement would always execute and would print "`b is true`" since the second `if` would evaluate to `True`.

6.28.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.28.5.
- Use either spaces or tabs, not both, to demark control flow.

6.29 Loop control variables [TEX]

6.29.1 Applicability to language

The vulnerabilities as documented in ISO/IEC 24772-1:2024 6.28 apply only minimally to Python. Python `for` loops iterate over structures such as lists or ranges.

It is possible to alter the loop behaviour by creating or deleting the objects that are iterated over. When using the `for` statement to iterate though an iterable object such as a list, there is no way to influence the loop count because it is not exposed. The variable `a` in the example below takes on the value of the first, then the second, then the third member of the list:

```
x = ['a', 'b', 'c']
for a in x:
    print(a)
#=>a
#=>b
#=>c
```

Python permits assignment expressions in loop control structures, that can result in either an endless loop, a prematurely terminated loop

It is possible, though not recommended, to change a mutable object as it is being traversed which in turn can change the number of iterations performed. In the case below the loop is performed only two times instead of the three times had the list been left intact:

Commented [SJM29]: Changing a mutable object does not necessarily change its length.

```
x = ['a', 'b', 'c']
for a in x:
    print(a)
    del x[0]
print(x)
#=> a
#=> c
#=> ['c']
```

6.29.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.29.5.
- Ensure to only modify variables involved in loop control in ways that are easily understood and do not result in unexpected behaviour, such as a premature exit or an endless loop.
- When using the `for` statement to iterate through a mutable object, avoid adding or deleting members because it could have unexpected results.
- Prohibit assignment expressions in the loop control statement (that is, `while` or `for`).

Commented [SJM30]: Do not result in unexpected behavior

Commented [SJM31]: What about scenarios such as:

```
def consumer(queue, id):
    print(f'consumer {id}: Running')
    while True:
        item = queue.get()
        if item is None
            queue.put(item)
            break
        sleep(item[1])
    print(f'\nconsumer {id}: Done')
```

6.30 Off-by-one error [XZH]

6.30.1 Applicability to language

The vulnerabilities described in ISO/IEC 24771-1 6.30 apply in part to Python.

The Python language itself is vulnerable to off-by-one errors as is any language when used carelessly or by a person not familiar with Python's index starting at zero versus at one. Python does not prevent off-by-one errors but its runtime bounds checking for strings and lists does lessen the chances that doing so will cause harm. It is also not possible to index past the end or beginning of a string or list by being off-by-one because Python does not use a sentinel character and it always checks indexes before attempting to index into strings and lists and raises an exception when their bounds are exceeded.

The `range` function can be used to create a sequence over a range of numbers such as:

```
for x in range(10):
     print (x)
```

which will print the numbers `0` through `9`. As many languages start indexing from `0`, this is not likely a source of great confusion. It is more likely that confusion will arise when using a range starting with a value other than the default 0, such as:

```
for x in range(5, 10):
     print (x)
```

which will print the values `5` through `9`.

6.30.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.30.5.
- Be aware of Python's indexing by default from zero and code accordingly.
- Be careful that a loop will always end when the loop index counter value is one less than the ending number of the range.
- Use the for statement to execute over whole constructs in preference to loops that index individual elements.
- Use the `enumerate()` built-in method when both container elements and their position within the iteration sequence are required.

6.31 Unstructured programming [EWD]

6.31.1 Applicability to language

The vulnerabilities described in ISO/IEC 24772-1:2019 6.31 are substantially mitigated in Python. The language does not provide a statement for local or non-local transfers of control; however there is a library that provides `goto` capabilities.

A `break` statement for the premature exit from loops is provided. Multiple `break` and multiple `return` statements are permitted. Breaking out of multiple nested loops from the innermost loop can be problematic as the `break` only terminates the nearest enclosing loop.

Python is designed to make it simpler to write structured program by requiring indentation to show scope of control in blocks of code:

```
a = 1
b = 1
if a == b:
    print("a == b") #=> a == b
    if a > b:
        print("a > b")
else:
    print("a != b")
```

In the example above, the indentation must be provided uniformly by the tab character or spaces. If tabs and spaces are mixed, the interpreter will reject the program.

In many languages the last `print` statement would be executed because the `else` is associated with the immediately prior `if` statement, while Python uses indentation to link the `else` with its associated `if` statement. In the example above, the `else` statement is associated with the first `if` statement since it has the same level of indentation.

Context managers (such as those introduced by the `with` keyword) can be used to consolidate where exceptions are evaluated and propagated, which lets developers write straight forward code without sprinkling `try` ... `except` ... `finally` structures throughout the code. For example, the following code ensures that the opened file is closed promptly, even if an exception occurs, or code in the body returns from a containing function, or breaks out of a containing loop:

```
with open("example.txt") as f:
    for line in f:
        print(line)
# File will be closed here,
# and on an exception, break, continue, or return
```

6.31.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.31.5.
- Avoid the use of the `goto` package.
- Use the `break` statement judiciously to exit from control structures and show statically that the code behaves correctly in all contexts.
- Restructure code so that the nested loops that are to be collectively exited form the body of a function, and use early function returns to exit the loops. This technique does not work if there is more complex logic that requires different levels of exit.
- Use context managers (such as `with`) to enclose code creating exceptions.

6.32 Passing parameters and return values [CSJ]

6.32.1 Applicability to language

The vulnerabilities as described in ISO/IEC TR 24772-1 6.32 minimally apply to Python.

Python functions return a value of `None` when no `return` statement is executed or when a `return` with no arguments is executed. Python detects attempts to return uninitialized arguments and raises the `NameError` exception.

Python passes arguments by assignment, which effectively is similar to passing by reference, as variables have references as their values. Python assigns the passed arguments to the function's local variables, but having the address of the caller's argument does not automatically allow the called function to change any of the objects referenced by those arguments as only `global` or mutable objects referenced by passed arguments can be changed. Aliasing can occur on the mutable objects designated by the parameters as follows:

```
class C():
    def __init__(self, number):
        self.comp = number

A=C(7)  # A.comp = 7
B=C(14) # B.comp = 14

def fun(X,Y):
   X.comp = 8
   Y.comp = 42
   print(X.comp) #=> may be 8, but also 42, depending on
call
   print(Y.comp) #=> always 42

fun(A, B) # call prints 8, 42
fun(A, A) # call prints 42, 42
fun(B, B) # call prints 42, 42
print(A.comp, B.comp) #=> 42 42
```

In the example above, class instances `A` and `B` are passed as arguments and their components are updated. While the local variables are discarded when the function

Commented [p32]: Suggest "which effectively is similar to passing by reference, as variables have references as their values."

Commented [SJM33R32]: OK

Commented [SM34R32]: Done

Commented [SJM35]: Or `global`

Commented [SM36R35]: Done

goes out of scope, changes to the components of their designated objects remain in effect. The example shows that when identical objects are passed as function arguments, e.g. `fun(A, A)` or `fun(B, B)`, the `X` and `Y` aliases in the function definition are reassigned with identical values and since `Y.comp` always appears after `X.comp`, its value always gets returned to the calling function.

The example below uses two class instances `A` and `B`, each passed individually into a function that uses the `B` class instance. When the class `B` instance is passed to the function, it is aliased to both internal variables `X` and `B`, but when class `A` is passed to the function, it is only aliased to `X`.

```
class C():
    def __init__(self, number):
        self.comp = number

def fun(X):
   X.comp = 9
   B.comp = 43
   print(X.comp) # may be 9, but also 43, depending on call
   print(B.comp) # always 43

A = C(7) # A.comp = 7
B = C(14) # B.comp = 14
fun(A) # call prints 9 43
fun(B) # call prints 43 43
```

In the example below, the argument is mutable, and is therefore updated in place:

```
a = [1]

def f(x):
    x[0] = 2
    if a[0] == 2:
        print("surprise!")

f(a) #=> surprise
print(a) #=> [2]
```

Note that the list object a is not changed – it is the same object but its content at index `0` has changed, which causes the aliasing effect demonstrated by the `if` statement.

Aliasing of arguments with immutable types cannot happen in Python. The following example demonstrates that one can emulate a call by reference by assigning the returned object to the passed argument:

```
def doubler(x):
    return x * 2
x = 1
x = doubler(x)
print(x) #=> 2
```

This is not a true call by reference and Python does not replace the value of the object `x`, rather it creates a new object `x` and assigns it the value returned from the doubler function as proven by the code below which displays the address of the initial and the new object x:

```
def doubler(x):
    return x * 2
x = 1
print(id(x)) #=> 506081728 changes with each execution
x = doubler(x)
print(id(x)) #=> 506081760 changes with each execution
```

The object replacement process demonstrated above follows Python's normal processing of any statement which changes the value of an immutable object and is not a special exception for function returns.

It is possible in Python to provide a read-only view of a parameter without the cost of making a local copy. The following example illustrates how to implement this read-only view by using the `MappingProxyType` interface:

```
from types import MappingProxyType
foo_types = MappingProxyType(
    {
        "foo1": 1,
        "foo2": 2
    }
)
print(foo_types["foo1"])
print(foo_types["foo2"])

#foo_types["foo1"] = 3 #=> TypeError: 'mappingproxy' object
                       #=> does not support item assignment
```

```
#=> OUTPUT
#=>  1
#=>  2
```

6.32.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.32.5 to avoid aliasing effects.
- Create copies of mutable objects before calling a function if changes are not wanted to mutable arguments.
- Use `types.MappingProxyType` or `collections.ChainMap` to provide read-only views of mappings without the cost of making a copy.
- Consider that local copies are created for immutable arguments when assignment occurs within the function, whereas for mutable arguments, assignments operate directly on the original argument.
- Be careful when passing mutable arguments into a function since the assignment sequence (order) within the function may produce unexpected results.

6.33 Dangling references to stack frames [DCM]

6.33.1 Applicability to language

With the exception of interfacing with other languages, Python does not have the vulnerability as described in ISO/IEC TR 24772-1 6.33. For example, Python has a foreign function library called `ctypes`, which allows C functions to be called in DLLs or shared libraries. It can provide the opportunity to read, and potentially change, arbitrary memory locations:

```
import ctypes
memid = (ctypes.c_char).from_address(0X0B98F706)
```

Once `memid` is known, the potential exists to modify the memory location.

See 6.53 Provision of inherently unsafe operations [SKL] for the avoidance of such inherently unsafe operations. For safer interactions with C code, Python provides the `cffi` module.

6.33.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

Commented [p37]: Justification missing in .1

Commented [SJM38R37]: Good catch!
We may want to delete this as an Avoidance Mechanism since it is tutorial in nature and we already have a significant number of examples in this section. However, if we do want to keep it and add justification in .1, here is an example usage:

```
from types import MappingProxyType

foo_types = MappingProxyType(
    {
        "foo1": 1,
        "foo2": 2
    }
)
print(foo_types["foo1"])
print(foo_types["foo2"])

#foo_types["foo1"] = 3 # => TypeError:
'mappingproxy' object does not support
item assignment

OUTPUT:
  1
  2
```

Commented [SM39R37]: Done

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.33.5.
- Avoid using `ctypes` when calling C code from within Python and use `cffi` (C Foreign Function Interface) instead.

6.34 Subprogram signature mismatch [OTR]

6.34.1 Applicability to language

The vulnerability of a mismatch in type expectations as described in ISO/IEC 24772-1:2024 6.34 exists in Python. An argument passed to a Python function may be of a type that does not match the needs of operations performed by the function on the formal parameter, resulting in a run-time exception. The other vulnerability of a mismatch in parameter numbers does not exist in Python, as Python checks the number of arguments passed. Variable numbers of positional and keyword arguments are supported by Python, but the method of accessing the arguments ensures that all access arguments exist.

Python supports the following argument structures:

1. positional,
2. `key=value` (called a keyword argument), or
3. both kinds of arguments, in which case positional arguments must precede the first keyword argument.

Python provides the mechanism `def foo(*a)` to permit `foo` to receive a variable number of positional arguments. In this case, the formal argument becomes a tuple and the actual parameters are extracted using tuple processing syntax. Furthermore, Python provides the mechanism `def foo(**a)` to permit `foo` to receive a variable number of keyword arguments called a dictionary.

Python always calls the most recently defined function of a specified name. That is, there is no overloading of arguments. There is no type-checking of arguments as part of parameter passing and no concept of function overloading. Type errors are detected when the body executes operations not available for the type of the argument. Python provides a type membership test `isinstance(var_name, Class_or_primitive_type)` that returns a Boolean that lets the user take alternative action based on the actual type of variable.

Python has many extension APIs and embedding APIs that include functions and classes providing additional functionality. These perform subprogram signature

checking at run time for modules coded in non-Python languages. Discussion of these APIs is beyond the scope of this document but the reader should be aware that improper coding of any non-Python modules or their interfaces can cause call stack problems. Programmers should also be aware that the `cffi` module will believe the signature information it is given, which may or may not be accurate. For vulnerabilities associated with calling libraries written in other languages, see 6.47 Inter-language calling [DIS].

6.34.2 Avoidance mechanisms for language users

To avoid the remaining vulnerability of type mismatches or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by described in ISO/IEC 24772-1:2024 6.47.5, Inter-language calling, when interfacing with C code or when calling library functions that interface with C code.
- Avoid using `ctypes` when calling C code from within Python; instead use the C Foreign Function Interface (`cffi`) since it is more streamlined and safer.
- Document the expected types of the formal parameters (type hints) and apply static analysis tools that check the program for correct usage of types.
- Use type membership tests to prevent runtime exceptions due to unexpected parameter types.

6.35 Recursion [GDL]

6.35.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.35 is mitigated in Python since the depth of the recursion is limited. Recursion is supported in Python and is, by default, limited to a depth of 1,000, which can be overridden using the `setrecursionlimit` function. If the limit is set high enough, a runaway recursion could exhaust all memory resources leading to a denial of service.

6.35.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.35.5.
- Use evidence when adjusting the maximum recursion depth to a larger value than the default

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.36 apply to Python.

Unhandled Python exceptions in the main thread will cause the program to terminate, as discussed in ISO/IEC 24772-1:2024 6.36.3. Unhandled exceptions in a concurrent part of a program will have effects that are dependent on the model of concurrency being used and the explicit way that the components are executed and communicate (see 6.62 Concurrency – Premature termination [CGS]).

The `assert` statement in Python is used primarily for debugging and throws an exception, with optional comment if the conditions of the assertion are not met. Such an exception must be handled to avoid terminating the program.

6.36.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.36.5.
- Ensure that all exceptions are caught and handled by appropriate handlers.
- Use the `assert` statement during the debugging phase of code development to help eliminate undesired conditions from occurring.
- Ensure that every exception that can be thrown is caught by the appropriate handler.

6.37 Type-breaking reinterpretation of data [AMV]

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.37 are not applicable to Python because assignments are made to objects and the object always holds the type, not the variable. Therefore, if multiple labels reference the same object, they all see the same type and there is no way to have more than one type for any given object.

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

Python exhibits the vulnerability as described in ISO/IEC 24772-1:2024 6.38.

Commented [p40]: Why is the paragraph here? Maybe better in precondition checking? Or in unexpected exceptions?

Commented [SJM41R40]: I agree with moving it to 6.36!

We currently have the following Avoidance *Mechanisms in 6.36.2:*

- *Use Python's exception handling mechanisms to ensure that only the desired named exceptions are caught and handled.*
- *Ensure that every exception that can be thrown is caught by the appropriate handler.*

Section **6.36 Ignored error status and unhandled exceptions** *may* be a good home for this sentence since the `assert` statement can be used to test the exceptions mentioned in these Avoidance Mechanisms (albeit typically for debugging only).

Commented [p42]: Ditto on placement

Commented [SM43R42]: OK?

The slice operator, e.g., "`x = y[:]`" and the copy methods, e.g. "`x = y.copy()`", copies the first level of a list, but leaves deeper levels, such as sub-lists, shared. For producing deep copies, Python provides the `deepcopy` method.

The following example illustrates the issues in Python:

```
colours1 = ["orange", "green"]
colours2 = colours1
print(colours1)               --  ['orange', 'green']
print(colours2)               --  ['orange', 'green']
colours2 = ["violet", "black"]
print(colours1)               --  ['orange', 'green']
print(colours2)               --  ['violet', 'black']
```

If, however, one writes:

```
colours1 = ["orange", "green"]
colours2 = colours1
colours2[1] = "yellow"
print(colours1)               --  ['orange', 'yellow']
```

When `colours1` is created, Python creates it as a list type, and then has the list point to its elements. When `colours2` is created as a copy of `colours1`, they both point to the same list container. If one sets a new value to an element of the list, then any variable that points to that list sees the update, as shown in the second example. The first example above shows that when a completely new list is created for `colours2` (replacing the equivalence of `colours1` and `colours2`), any further changes to `colours2` or colours1 do not affect the other.

Copying with the slice operator `[:]` provides a deeper level of copying under certain situations. It does create a new memory address for the top-level list, but when embedded sublists are involved, the slice operator still references the objects in the original list. The following example shows how changing a sublist within list L2 also unintentionally changes the same sublist in list L1.

```
L1 = [[1,2,3], [4,5,6], [7,8,9]]
L2 = L1[:]
L2[0][2] = [123456789]
print(L1) #=> [[1, 2, [123456789]], [4, 5, 6], [7, 8, 9]]
print(L2) #=> [[1, 2, [123456789]], [4, 5, 6], [7, 8, 9]]
```

Python also has a function called `deepcopy` that can be imported from the `copy` module and copies all levels of a structured object to a completely new object so that a list within a list can be independently accessed as shown in the example below:

```
import copy
L1 = [[1,2,3], [4,5,6], [7,8,9]]
L2 = copy.deepcopy(L1)
L2[0][2] = [123456789]
print(L1) #=> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(L2) #=> [[1, 2, [123456789]], [4, 5, 6], [7, 8, 9]]
```

6.38.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.38.5.
- Be aware that the slice operator `[:]` and the container `copy` methods only perform shallow copies.
- Use the `copy.deepcopy` standard library function to obtain deep copies at all levels of a variable.

6.39 Memory leaks and heap fragmentation [XYL]

6.39.1 Applicability to language

The heap fragmentation vulnerability as described in ISO/IEC 24772-1:2024 6.39 exists in Python. The memory leak vulnerability of that subclause is mitigated by Python automatic garbage collection as described below.

Python supports automatic garbage collection so in theory it should not have memory leaks. However, there are at least three general cases in which memory can be retained after it is no longer needed.

The first case is when implementation-dependent memory allocation/de-allocation algorithms cause a leak, which would be an implementation error and not a language error.

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.

The third case is a subtle memory leak case wherein objects mutually reference one another without any outside references remaining – a kind of deadly embrace where one object references a second object (or group of objects) so the second object (or

group of objects) cannot be collected but the second object(s) also reference the first one(s) so it/they too cannot be collected. This group is known as cyclic garbage. Python provides a garbage collection module called `gc` which has functions which enable the programmer to enable and disable cyclic garbage collection as well as inspect the state of objects tracked by the cyclic garbage collector so that these, often very subtle leaks, can be traced and eliminated.

6.39.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.39.5.
- Set each object to null when it is no longer required.
- For programs intended for continuous operation, examine all object usage carefully, applying the avoidance mechanisms provided by ISO/IEC 24772-1, to show that memory is effectively reclaimed and reused.
- Use context managers to explicitly release large memory buffers that are no longer needed.

6.40 Templates and generics [SYM]

6.40.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.40 apply to Python, although Python does not have the applicable language characteristics as outlined in ISO/IEC 24772-1:2024 6.40.4. Since Python is dynamically typed, essentially all functions in Python exhibit generic properties. Therefore, the mechanisms of failure outlined in ISO/IEC 24772-1:2024 6.40.3 apply to Python.

6.40.2 Avoidance mechanisms for language users

Software developers can avoid the vulnerabilities or mitigate their ill effects by applying the avoidance mechanisms of ISO/IEC 24772-1:2024 6.40.5.

6.41 Inheritance [RIP]

6.41.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.41 apply to Python.

Python supports inheritance as described in 5.1.6.

It is important to make sure that each class calls the `__init__` of its superclass so that it is properly initialized. The built-in function `super()` provides access to the next class in the MRO sequence. See 5.1.6, which also includes an example.

The difficulties associated with establishing the MRO are also illustrated in 5.1.4.

There can be unexpected outcomes from the MRO as shown in the following code. The outcome might be expected to be `a=0`, but in reality, the result is `a=2` since, as previously mentioned, methods in derived calls are always called before the method of the base class (`class T`).

```
class T():
    a = 0
class A(T):
    pass
class B(T):
    a = 2
class C(A,B):
    pass
c = C()
print(c.a) # => 2
```

There is no protection in Python against accidental redefinition, method capture, or accidental non-redefinition along the MRO sequence, so that these vulnerabilities apply.

Moreover, as the search for a binding is at run-time in dynamically established class hierarchies , a static analysis cannot predetermine the danger of these vulnerabilities to incur. Neither can a reviewer of the code without detailed analysis of the entire class hierarchy determine which method is called. The `__mro__` attribute can be queried in the code to determine the MRO sequence.

Hailed as a flexibility in Python literature, it is possible to add an additional sibling class into a given hierarchy, thereby redefining parent method definitions (or adding new ones), so that the elder sibling appears to have these capabilities from the viewpoint of all classes below. Thus, incorrect or malicious code can be inserted into already validated code.

As explained in 5.1.4 Mutable and Immutable Objects , there are situations in which Python cannot establish a consistent MRO, in which case the `TypeError` exception is

raised. For a discussion of vulnerabilities related to unhandled exceptions, see 6.36 Ignored error status and unhandled exceptions [OYB].

There are no language mechanisms to enforce class invariants when methods are redefined, so that class invariants can be easily violated by redefinitions.

To enforce the use of getter and setter methods to access class members, Python provides a mechanism to make members effectively private: the use of leading double underscores (without matching trailing underscores) for their name implies only local visibility in Python.

Any inherited methods are subject to the same vulnerabilities that occur whenever using code that is not well understood.

Static type analysis tools can detect issues associated with complex class hierarchies. Python's type hints provide valuable information to static analysis tools. Similarly, in multiple inheritance situations, displaying the MRO sequence assists developers in understanding the method binding (see 6.44 Polymorphic variables [BKK]).

6.41.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.41.5.
- Inherit only from trusted classes, such as standard classes.
- Only use multiple inheritance that is linearizable by the MRO rules.
- Make sure that each class calls the `__init__` of its superclass.
- Use the `__mro__` attribute to obtain information about the MRO sequence of classes followed by method calls.
- Use static analysis tools supported by type-checking hints.
- Employ type hints to elicit compile-time analysis.
- Prefix method calls with the desired class wherever feasible.
- Use Python's built-in documentation (such as docstrings) to obtain information about a class' methods before inheriting from the class.
- For users who are new to the use of multiple inheritance in Python, carefully review Python's rules, especially those of `super()` and class names that prefix calls.

6.42 Violations of the Liskov substitution principle or the contract model [BLP]

6.42.1 Applicability to language

Python is subject to violations of the Liskov substitution rule as documented in ISO/IEC 24772-1:2024 6.42. The Python community provides static analysis tools for Python, which detect some violations of the Liskov Substitution Principle, such as on arguments and results of methods of subclasses.

6.42.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.42.5.
- Use software static analysis tools to help identify violations.

6.43 Redispatching [PPH]

6.43.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.43 exist in Python. By default, all calls in Python resolve to the method of the controlling object, a semantics that ISO/IEC 24772-1:2024 refers to as redispatching, and thus can result in infinite recursion between redefined and inherited methods, as described in ISO/IEC 24772-1.

Redispatching can be prevented by:

- Prefixing the method call by the name of the desired class; or
- Prefixing the method call by `super()` to call on the method found along the MRO of the current class.

The following example shows the infinitely recursive dispatching caused in `h()` and prevented in `f()`:

```
class A:
  def f(self):
    print("In A.f()”)
```

Commented [p44]: I VERY MUCH doubt this. How can you possibly distinguish automatically “is-a” and “has-a” relationships?

Commented [SJM45R44]: Ref: https://github.com/python/typing/issues/487

According to Guido, in response to Stephen’s question on the topic:

“… the mypy checker does mitigate this by flagging Liskov violations as errors…”

Also
“mypy type checker detects and prohibits Liskov violations with very few exceptions (like incompatible __init__ method signatures). ”

Commented [SM46R44]: OK

Commented [p47]: Ditto

Commented [SJM48R47]: The following screenshot illustrates mypy in use and how it finds Liskov violations in the example found in: https://mypy.readthedocs.io/en/stable/error_code_list.html#check-validity-of-overrides-override

… [2]

Commented [SM49R47]: OK

```
  def g(self):
    A.f(self) # call to f() in subclass B, will not dispatch
  def h(self):
    self.i()
  def i(self):
    self.h() # call to h() in subclass B, will dispatch
             # showing the vulnerability
class B(A):
  def f(self):
    self.g()
  def h(self):
    self.i() # call to i() in superclass A (infinite
recursion)

a = A()
b = B()
b.f() #=> In A.f()
b.h() #=> RecursionError: maximum recursion depth exceeded
```

An important consideration in `class` definition is that Python permits a second method in a `class` with identical signature to an earlier one, which effectively hides the first one and prevents it from being called.

See 6.44 Polymorphic variables [BKK] for associated vulnerabilities.

6.43.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.43.5.
- Avoid dispatching whenever possible by prefixing the method call with the target class name, or with `super()`.
- Within a single class, avoid the definition of a second method with the same signature as an existing method.
- Use systematic code reviews, organization-wide coding standards, and static analysis tools to prevent problems related to the redefinition of methods in object-oriented programming.

Commented [p50]: Is this legal at all? Or is this "within a class hierarchy"?

Commented [SJM51R50]: Yes, this is legal:

```
class fooclass():

    def foo(self):
        print("in first foo")

    def foo(self):
        print("in second foo")

f = fooclass()
print(f.foo())

OUTPUT:
in second foo
None
```

Same behavior occurs outside of a class:

```
def foo():
    print("in first foo")
def foo():
    print("in second foo")
print(foo())
OUTPUT:
in second foo
None
```

Commented [SM52R50]: See my proposed explanation in 6.43.1

6.44 Polymorphic variables [BKK]

6.44.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.44 exist in Python in principle, although the mechanisms differ from the ones described in ISO/IEC 24772-1.

Python is inherently polymorphic, in the sense that any called operation will attempt to apply itself to the given object and raise an exception if it cannot apply the operation (see 5.1.6 Inheritance).

While Python has no casting operators as described in ISO/IEC 24772-1:2024, prefixing method calls with class names can achieve similar effects for these calls and cause respective vulnerabilities:

- Prefixing a call with the name of a specific class forces the binding of the method name to be taken from this class. There is, however, no check performed whether the named class is an ancestor class of the class of the `self` object, and thus safe to use (commonly known as "upcast"). Any class is accepted, turning the feature into an unsafe cast in the terminology of ISO/IEC 24772-1. Subsequent failures occur in Python only when the class of self does not have members named by the implementation of the chosen method, or, if it does, malfunctions arise when the user semantics of these members are different in the two classes, e.g., a member count in two unrelated classes may stand for the count of very different entities, a method engage may engage an engine or engage a loving couple, depending on the class involved. Since parameters play no role in method resolution, they do not help in avoiding unintended matches.
- "`super()`" as a prefix to a call ignores local definitions and, instead, picks the binding from the next class in the applicable MRO (often a parent class as in most OO-languages, but occasionally a sibling of the parent class, as shown in the example in 5.1.6). As such, it is reasonably safe, since the classes are ancestors of the class of the object, albeit possibly not yielding the expected binding. The vulnerabilities of upcasts, as described in ISO/IEC 24772-1, apply in any case. The `super()` function returns a temporary proxy object of the superclass so that its name does not need to be used in the child class. The example below shows how to explicitly call the `__init__` method in the `Foo` superclass by using both the

superclass name and the `super()` function. Notice that the self-object reference parameter is required when using the `Foo` superclass name. Notice also that, by using `super()`, any changes to the parent class name will not matter as they do for the first call.

```
class Foo(object):
    def __init__(self, msg):
        print(msg)

class DerivedFoo(Foo):
    def __init__(self)
        Foo.__init__(self, '__init__ using Foo')
                                    # => __init__ using Foo
        super().__init__('__init__ using super()')
                                    # => __init__ using
super()
DerivedFoo()
```

Commented [p53]: True, but is it worth mentioning?

Commented [SJM54R53]: Somewhat tutorial however the concept of `super()` is somewhat unique to Python.

Commented [SM55R53]: OK

6.44.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.44.5.
- Ensure that each class implements the `__init__` method that calls the `__init__` of its superclass.
- Employ static type checking by providing type hints for static analysis tools in areas involving inheritance.
- Use `__mro__` as an aid during development and during maintenance to help obtain the desired class hierarchies and verify linearity.
- Consider using `__mro__` to check at runtime that the actual method binding matches the expected method binding and to raise an exception if they do not match.
- Pay attention to warnings that identify variables written but never read.

6.45 Extra intrinsics [LRM]

6.45.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1:2024 6.45 applies to Python.

Python provides a set of built-in intrinsics, which are implicitly imported into all Python scripts. Any of the built-in variables and functions can therefore easily be overridden as in this example:

```
x = 'abc'
print(len(x)) #=> 3
def len(x):
    return 10
print(len(x)) #=> 10
```

In the example above the built-in `len` function is overridden with logic that always returns `10`. Note that the `def` statement is executed dynamically so the new overriding `len` function has not yet been defined when the first call to len is made therefore the built-in version of `len` is called in line 2 and it returns the expected result (`3` in this case). After the new `len` function is defined it overrides all references to the builtin-in `len` function in the script. This can later be "undone" by explicitly importing the built-in `len` function with the following code:

```
from builtins import len
print(len(x)) #=> 3
```

It is very important to be aware of name resolution rules when overriding built-ins (or anything else for that matter). In the example below, the overriding `len` function is defined within another function and therefore is not found using the LEGB rule for name resolution (see 6.21 Namespace issues [BJL]):

```
x = 'abc'
print(len(x)) #=> 3
def f(x):
    def len(x):
        return 10
print(len(x)) #=> 3
```

6.45.2 Avoidance mechanisms for to language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.45.5.
- Prohibit the overriding of built-in intrinsics.

- If it is necessary to override an intrinsic, document the case and show that it behaves as documented and that it preserves all the properties of the built-in intrinsic.

6.46 Argument passing to library functions [TRJ]

6.46.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1:2024 6.46 applies to Python.

6.46.2 Avoidance mechanisms for language users

Software developers can avoid the vulnerability or mitigate its ill effects by applying the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.46.5.

6.47 Inter-language calling [DJS]

6.47.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.47 are mitigated in Python, which has documented API's for interfacing with other languages. Python has an API that extends Python using libraries coded in C or C++. The library or libraries are then imported into a Python module and used in the same manner as a module written in Python. The full API exposed to the "C" language by the CPython reference interpreter is documented in the "Python/C API Reference Manual"[14]. The section in the Python/C API Reference Manual entitled "Extending Python with C or C++" provides a low-level example of writing an extension module from scratch using that API.

Conversely, code written in C or C++ can embed Python. The standard for embedding Python is documented in "Embedding Python in Another Application" [3].

Writing Python extension modules by hand is error-prone, and highly likely to lead to reference counting errors, memory leaks, dangling pointers, out-of-bounds memory accesses, and similar problems.

Note that Python maintainers recommend that developers use existing libraries and tools that automatically generate the Python interface code from simpler descriptions of intent, such as those covered in Packaging binary extensions [9] such as `Cython`, `cffi`, and `SWIG`. Other libraries that can be used for performance optimization are `PyO3` for Rust, and `pybind11` for C++.

6.47.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 47.5, especially when interfacing to a language without a predefined API.
- Avoid writing Python extension modules by hand.
- Where available, use existing interface libraries that bridge between Python and the extension module language,

6.48 Dynamically-linked code and self-modifying code [NYY]

6.48.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.48 apply to Python.

Python supports dynamic linking by design. The `import` statement fetches a file (known as a module in Python), compiles it and executes the resultant byte code at run time. This is the normal way in which external logic is made accessible to a Python program. Therefore, Python 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 to locate a different file first.
- Overlaying of a file with an alternate file.

Python also provides the `eval` and `exec` statements. The `exec` statement compiles and executes statements (example: `x=1`, a line that requires execution). The `eval` statement evaluates expressions (example, `1+1`, composed of operators and expressions). Both statements can be used to create self-modifying code:

```
x = "print('Hello ' + 'World')"
eval(x)          #=> Hello World
program = \
"a = 5"\
"b = 10"\
print("Sum =", a+b)"
exec(program)    # Output: Sum = 15
```

Guerrilla patching, also known as monkey patching, is a way to dynamically modify a module or class at run-time to extend or subvert their processing logic and/or attributes. It can be a dangerous practice because once "patched" any other modules or classes that use the modified class or module may unwittingly use code that does not do what is expected, which could cause unexpected results.

Python, by default, is liable to execute dangerous code without detection or verification. The Python interpreter provides a default entry point that allows execution with no hooks enabled. Production software that uses modified entry points and logs as many events as possible can reduce most of these risks.

Python Enhancement Proposal (PEP) 578 [12] documents issues with audit hooks as using them can alter the behaviour of runtime calls and provides advice to eliminate their default behaviour.

6.48.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.48.5.
- Avoid using `exec` or `eval` and never use these with untrusted code.
- Avoid guerrilla patching,
- If guerrilla patching is unavoidable, ensure that all uses of the patched classes and/or modules continue to function as documented through mechanisms such as audit hooks and event logging.
- Use caution when including any code that patches classes and/or modules.
- Ensure that any file paths and files being imported are from trusted sources.
- Consider the guidance of PEP 578 [12] and its predecessor PEP 551 [11] to eliminate potentially dangerous default behaviour from calls into the Python runtime and in the use of audit hooks.
- Verify that the release version of the product does not use default Python entry points (`python.exe` on Windows, and `pythonX.Y` on other platforms) since these are executable from the command line and do not have hooks enabled by default.
- Consider using a modified entry point that restricts the use of optional arguments to reduce the chance of unintentional code being executed in place of the default entry point.
- Avoid unprotected settings from the working environment in entry points.
- If the application is performing event logging as part of normal operations, consider logging all predetermined events in calling external libraries.

- Consider logging as many events as possible and ensure that such logs are archived to an external location.

6.49 Library signature [NSQ]

6.49.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.49 are mitigated in Python, which provides an extensive API for extending or embedding Python using modules written in C, Java, and Fortran. Extensions themselves have the potential for vulnerabilities exposed by the language used to code the extension, which is beyond the scope of this document.

Python does not have a library signature-checking mechanism, but its API provides functions and classes to help ensure that the signature of the extension matches the expected call arguments and types (see 6.34 Subprogram signature mismatch [OTR]).

Python does provide an API that gives access to various runtime, import and compiler events. The information gathered from these events can be used to detect, identify and avoid malicious activity. For example, `sys.audithook` can be used to add a callback function for a predefined set of events. The callback function receives the name of the event as well as arguments that can be used for monitoring and filtering. These monitored events can be used to evaluate third party components for suspicious activity during runtime, reducing the inherent risks associated with external modules. These hooks are useful in situations where third-party source code is either unavailable or too large to evaluate for malicious activity.

6.49.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.49.5.
- Use only trusted modules as extensions.
- If coding an extension, utilize Python's extension API to ensure a correct signature match.

6.50 Unanticipated exceptions from library routines [HJW]

6.50.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.50 applies to Python.

Python is often extended by importing modules coded in Python and other languages. For modules coded in Python, the risks include the interception of an exception that was intended for a module's imported exception handling code and vice versa.

For modules coded in other languages, the risks include:

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

6.50.2 Avoidance mechanisms for language users

Software developers can avoid the vulnerability or mitigate its ill effects by applying the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.50.5.

6.51 Pre-processor directives [NMP]

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.51 do not apply to Python since Python does not have a preprocessor.

6.52 Suppression of language-defined run-time checking [MXB]

6.52.1 Applicability to language

The vulnerabilities as documented in ISO/IEC 24772-1:2024 6.52 apply to Python.

Among the mechanisms to suppress runtime checking or reporting of runtime errors are:

- Using the command line option specific to the execution environment;
- Using the `catch_warnings` function to catch and subsequently ignore warnings;
- Catching and then ignoring runtime exceptions.

Each of these mechanisms provide ways that serious situations that are detected by the runtime can be ignored, which will almost always result in significant vulnerabilities.

6.52.2Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Follow the avoidance mechanisms or ISO IEC 24772-1 6.52.5.
- Forbid suppressing runtime checks.
- Forbid ignoring caught warnings.
- Forbid ignoring caught runtime exceptions.

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.53 apply to Python.

Even though there is no way to suppress error checking or bounds checking in Python, there are features that are inherently unsafe:

- Interfaces to modules coded in other languages since they could easily violate the security of the calling of embedded Python code (see 6.47 Inter-language calling [DJS]).
- Use of the `exec` and `eval` dynamic execution functions (see 6.48 Dynamically-linked code and self-modifying code).
- Similarly, `logging.dictConfig` can end up running arbitrary code.
- Python permits user-defined modifications of the contents of module builtins. Doing so, however, can be unsafe unless the redefinition matches all of the semantics of the original built-in function, including future enhancements. Overriding Python's default behaviour, by either overriding Python's built-in functions or hiding it or a built-in variable by a user-defined variable of the same name, can have undesired side effects and can be difficult to debug.
- The `pickle` module is inherently unsafe since it allows arbitrary, and potentially malicious, code execution. `pickle` can spawn anything that Python can invoke including the web browser. To mitigate this risk, whitelists of Python built-in functions that are deemed to be expected and acceptable can be created, and all other functions disallowed.

- Older Python 2 `pickle` protocols can be ASCII and slow (protocol=0) making them especially prone to DOS attacks. Python 3 defaults to higher protocols (2-4, binary). The anticipated protocol to be used is determined when pickled, but an attacker can choose various protocols. This risk can be reduced by not using `protocol 0`.
- `pickle` bombs (self-referencing payloads) can make a small payload expand to an extremely large object in memory resulting in DOS or other attacks. There are legitimate use cases for self-referencing payloads, but in order to minimize the chance of them being misused and potentially leading to a DOS attack, self-referencing payloads can be disallowed.
- Usage of `pickle` for long-term storage increases the risk of attack, due in part to many more `pickle` payloads that are accepted than generated, and to evolving protocol and Python version changes.

6.53.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.53.5.
- Use only trusted modules.
- Avoid the use of the `exec` and `eval` functions.
- Avoid overriding Python's default behaviour provided by the builtins module.
- Create a whitelist of Python built-in functions that are deemed to be expected and acceptable in uses of `pickle` and forbid any other functions.
- Forbid overriding the names of built-in variables or functions.
- Avoid the use of the `pickle` module and `logging.dictConfig` and consider using `JSON` and `MessagePack` as alternatives.
- Avoid the use of `pickle` for long term storage.
- Avoid the use of `protocol 0`.
- Disallow the use of self-referencing payloads.

6.54 Obscure language features [BRS]

6.54.1 Applicability of language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.54 apply to Python. Some examples of obscure language features in Python are:

- Functions are defined when executed:

```
a = 1
```

```
while a < 3:
      if a == 1:
         def f():
             print("a must equal 1")
      else:
         def f():
             print("a must not equal 1")
      f()
      a += 1
```

The function f is defined and redefined to result in the output below:

```
a must equal 1
a must not equal 1
```

- A function's variables are determined to be local or global using static analysis: if a function only references a variable and never assigns a value to it then it is assumed to be global otherwise it is assumed to be local and is added to the function's namespace. This is covered in some detail in 6.22 Initialization of variables [LAV].
- A function's default arguments are assigned when a function is defined, not when it is executed:

```
def f(a=1, b=[]):
    print(a, b)
    a += 1
    b.append("x")
f()
f()
f()
```

The output from above is typically expected to be:

```
1 []
1 []
1 []
```

But instead, it prints:

```
1 []
1 ['x']
1 ['x', 'x']
```

This is because neither `a` nor `b` are reassigned when `f` is called with no arguments because they were assigned values when the function was defined. The local variable `a` references an immutable object (an integer) so a new object is created when the `a += 1` statement is executed and the default value for the `a` argument remains unchanged. The mutable list object `b` is updated in place and thus is extended with each new call.

- The += operator does not work as might be expected for mutable objects:

```
x = 1
x += 1
print(x) #=> 2 (Works as expected)
```

But when we perform this with a mutable object:

```
x = [1, 2, 3]
y = x
print(id(x), id(y)) #=> 38879880 38879880
x += [4]
print(id(x), id(y)) #=> 38879880 38879880
x = x + [5]
print(id(x), id(y)) #=> 48683400 38879880
print(x, y) #=> [1, 2, 3, 4, 5] [1, 2, 3, 4]
```

- The += operator changes x in place while the x = x + [5] creates a new list object which, as the example above shows, is not the same list object that y still references. This is Python's normal handling for all assignments (immutable or mutable) – create a new object and assign to it the value created by evaluating the expression on the right-hand side (RHS):

```
x = 1
print(id(x)) #=> 506081728
x = x + 1
print(id(x)) #=> 506081760
```

- Equality (or equivalence) refers to two or more objects having the same value. It is tested using the == operator which compares values. On the other hand, two or more names in Python are considered identical only if they reference the same object which can be tested by using the `is` keyword (in which case they would, of course, be equivalent too). For example:

```
a = [0,1]
b = a
c = [0,1]
```

```
a is b, b is c, a == c #=> (True, False, True)
```

`a` and `b` are both names that reference the same objects while `c` references a different object which has the same value as both `a` and `b`.

- Python's `pickle` module provides built-in classes for persisting objects to external storage for retrieval later. The complete object, including its methods, is serialized to a file (or DBMS) and re-instantiated at a later time by any program which has access to that file/DBMS. This has the potential for introducing rogue logic in the form of object methods within a substituted file or DBMS.
- Python supports defaults for function parameters, as in:

```
def f(a=1, b=[]):
   print(a, b)
   a += 1
   b.append("x")_
f() # => 1 []_
f() # => 1 ['x']
f() # => 1 ['x', 'x']
```

However, using mutable default parameters can cause surprising effects since Python's default arguments are evaluated only once when the function is defined, not each time the function is called.

- Python has functions as first class objects that can be passed as arguments, which can be confusing in the wrong context. For example, the following two function calls

```
myFunc(target=doIt)
```

and

```
myFunc(target=doIt())
```

have different semantics. In the first case, the function doIt is passed as an argument and can be called from within myFunc; in the second case, the result of calling the `doIt()` function is passed as the argument. It is important that readers of the code be aware of the major semantic difference caused by adding the argument list.

6.54.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.54.5.
- 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.
- Be aware of when a variable is local versus `global`.
- Avoid mutable objects as default values for arguments in a function definition unless absolutely needed and the effect is understood.
- Be aware that when using the += operator on mutable objects the operation is done in place with a new object not being created.
- Be cognizant that assignments to objects, mutable and immutable, always create a new object.
- Be aware of the syntactic difference between a function name and a function call without arguments.
- Understand the difference between equivalence and equality and code accordingly.
- Ensure that the file path used to locate a persisted file or DBMS is correct and never ingest objects from an untrusted source.

6.55 Unspecified behaviour [BQF]

6.55.1 Applicability of language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.55 apply to Python to a limited extent, as follows:

- The sequence of keys in a set is unspecified because the hashing function used to index the keys is likely to yield different sequences depending on the implementation.
- Python sets are unordered and unindexed, thus cannot be sorted. Any attempt to sort them has unspecified behaviour. In addition, other functions that depend on order, such as `min()`, `max()`, and `sorted()` have unspecified behaviour over sets.
- When creating persisting objects, if an exception is raised then an unspecified number of bytes may have already been written to the file.
- Relying on Python's garbage collector to destroy a pool will not guarantee that the finalizer of the pool will be called.

Commented [p56]: Duplicate this to unspecified behavior. Strange, though, to read this specifically for Threadpools. What about other data structures: Is it the case that finalizers for anything might not be called by GC in general?

Commented [SJM57R56]: Also applicable to processes

- Pickling can result in unspecified behaviour as documented in 6.53.1 Provision of inherently unsafe operations [SKL].
- For integers within the range [-5:256], Python optimizes duplicate assignments but, for all other values, each replicated variable points to its own unique object:

```
a = 256
b = 256
print(a is b) #=> True
a = 257
b = 257
print(a is b)  #=> False
```

6.55.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 55.5.
- When pickling is applied to make objects persistent, use exception handling to cleanup partially written files.
- Be aware of the difference between equality (`==`) and identity (`is`) and use them as appropriate.
- Use the `intern()` function to enforce optimization when memory optimization is required for non-simple strings.
- Consider using the `id()` function to test for object equality.
- Finalize all pools before destroying them.
- Forbid form feed characters for indentation.

6.56 Undefined behaviour [EWF]

6.56.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.56 apply to Python. Python has undefined behaviour in the following instances, among others:

- The behaviour of the Future class encapsulating the asynchronous execution of a callable is undefined if the `add_done_callback(fn)` method (which attaches the callable `fn` to the future) raises a `BaseException` exception.

- Modifying the dictionary returned by the `vars()` and `locals()` built-ins have undefined effects when used to retrieve the dictionary (that is, the namespace) for an object. The `vars()` built-in can accept an optional object as a parameter `vars(obj)` and, in this case, the returned value is not undefined but depends on the type of the parameter object.
- The `catch_warnings` function in the context manager can be used to temporarily suppress warning messages but it can only be guaranteed in a single-threaded application; otherwise, when two or more threads are active, the behaviour is undefined.
- When sorting a list using the `sort()` method, attempting to inspect or mutate the content of the list will result in undefined behaviour.
- Undefined behaviour will occur if a thread exits before the main procedure, from which it was called.

6.56.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.56.5.
- Ensure that a callable does not raise a `BaseException` if launched as a parallel task using the `add_done_callback(fn)` command.
- Avoid dependence on the consistencies of the sequence of keys in a dictionary across implementations, or even between multiple executions with the same implementation, in versions prior to Python 3.7.
- Forbid modification of the dictionary object returned by a `vars()` and `locals()` call.
- Forbid the use of the `catch_warnings` function to suppress warning messages when using more than one thread.
- Forbid inspecting or changing the content of a list when sorting a list using the `sort()` method.

6.57 Implementation–defined behaviour [FAB]

6.57.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.57 apply to Python. For example, Python has implementation-defined behaviour in the following instances:

- Byte order (little endian or big endian) varies by platform.
- Exit return codes are handled differently by different operating systems.

- The characteristics of floating-point types, such as the maximum number of decimal digits that can be represented, vary by platform.
- The filename encoding used to translate Unicode names into the platform's filenames varies by platform.
- Python supports integers whose size is limited only by the memory available on the platform. Extensive arithmetic using integers larger than the largest integer supported on the platform used to implement Python will degrade performance.
- The type of garbage collection algorithm used, such as "reference counting" or "mark and sweep", is implementation-defined. Depending upon the algorithm used, additional programmer action is required to prevent memory leakage.
- The maximum value that a variable of type `Py_ssize_t` can take is implementation defined and documented by `sys.maxsize`.
- Python uses string interning which is a process of storing only one copy of each distinct string value (up to 4096 characters in length) in memory. For efficiency reasons, whether a string will be interned and the interning mechanism that Python uses for strings and integers varies depending on object characteristics. For example, when a copy of a string that meets certain characteristics is created in Python, the copy points to the same object as the original:

```
a = 'StringWithOnlyASCIILetters_Digits_And_Underscores'
b = 'StringWithOnlyASCIILetters_Digits_And_Underscores'
print(a == b, a is b) #=> True True
```

All other strings, such as those longer than 4096 characters or containing any character that is not an ASCII letter, digit, or underscore, will not be interned.

```
a = 'Non-Simple String!' # ' ' and '!' prevent this
                         # string from being interned
b = 'Non-Simple String!'
print(a == b, a is b) #=> True False
```

Note the unexpected `False` in the result.

If memory optimization is required for non-simple strings, it can be enforced by using the `intern()` function:

```
from sys import intern
a = intern('Non-Simple String!')
```

Commented [p58]: On the platform! Otherwise a direct contradiction.

Commented [SJM59R58]: Good point! Agree, change from language to *platform*.

Commented [SM60R58]: Done.

```
b = intern('Non-Simple String!')
print(a == b, a is b) #=> True True
```

In general, executions of a program initiated in different ways, such as from the command line or from invocation by another program, can result in different outcomes due to implementation-defined elements in Python.

6.57.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.57.5.
- Either avoid logic that depends on byte order or test the `sys.byteorder` variable and write the program logic to account for byte order `little` or `big`.
- Use `zero` (the default exit code for Python) for successful execution and consider adding logic to vary the exit code according to the platform as obtained from `sys.platform` (such as, '`win32`', '`darwin`', or other).
- Interrogate the `sys.float.info` system variable to obtain platform specific attributes and code according to those constraints.
- Call the `sys.getfilesystemcoding()` function to return the name of the encoding system used.
- Use the `os.fsencode()` and `os.fsdecode()` methods as a portable way to encode or decode a filename to the filesystem encoding that is used.
- When high performance is dependent on knowing the range of integer numbers that can be used without degrading performance use the `sys.int_info` struct sequence to obtain the number of bits per digit (bits_per_digit) and the number of bytes used to represent a digit (`sizeof_digit`).
- Use `sys.maxsize` to determine the maximum value a variable of type `Py_ssize_t` can take. Usually on a 32-bit platform, the value is 2**31 - 1 on a 32-bit platform and 2**63 - 1 on a 64-bit platform.
- When portable code is required, always execute on several different Python implementations and different invocation methods.

6.58 Deprecated language features [MEM]

6.58.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.58 apply to Python. For example, the following features were deprecated in Python:

- The `string.maketrans()` function is deprecated and is replaced by new static methods, `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the string module. Now, str, bytes, and bytearray each have their own `maketrans()` and translate methods with intermediate translation tables of the appropriate type.
- The syntax of the `with` statement now allows multiple context managers in a single statement:

```
with open('mylog.txt') as infile, open('a.out', 'w') as
outfile:
    for line in infile:
         if '<critical>' in line:
             outfile.write(line)
```

  With the new syntax, the `contextlib.nested()` function is no longer needed and is now deprecated.

- `PyNumber_Int()` is deprecated. Use `PyNumber_Long()` instead.
- The functions `PyOS_ascii_strtod()` and `PyOS_ascii_atof()` are deprecated and have been replaced by function `PyOS_string_to_double()`.
- The `PyCObject` API has been deprecated and replaced by `PyCapsule`, which has a well-defined interface for passing typing safety information and a less complicated signature for calling a destructor.

Warnings resulting from `DeprecationWarning` are shown by default but only when triggered by code running in the `__main__` module.

6.58.2 Avoidance mechanism for language users

Software developers can avoid the vulnerabilities or mitigate their ill effects by applying the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.58.

6.59 Concurrency – Activation [CGA]

6.59.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1 6.59 apply to Python.

Python provides multiple concurrency models (see 5.1.7 Concurrency).

Threading model

When a thread is created, if the new thread fails to be created for any reason, then an exception is thrown in the execution path of the creator, which can take corrective action. Hence this vulnerability does not exist for Python threads.

On the other hand, if a child thread has already been started, then attempting to start it again will result in an exception, and the behaviour of the program is implementation-defined. This applies even if the started thread has completed.

This scenario can lead to deadlock and race conditions when activating a thread, and is not always observable even during extensive testing, so it is important to prevent it during development so that it does not surface later.

The `ThreadPoolExecutor` enables a predetermined number of threads to be created in advance and available for work. Otherwise, creating and then destroying threads in Python has significant overhead associated with it so keeping a pool of threads available eliminates the creation/destruction process. The `join()`operation is also performed automatically so that is another benefit.

Multiprocessing model

Since the processing model used is that of the underlying operating system and all process interactions are those of the OS, the vulnerabilities are those of the underlying OS.

Calling `set_start_method()` more than once on the same child process causes an exception. Calling it conditionally, for example with the `if __name__ == '__main__'` statement ensures that a process can be started only by a module called `'__main__'`.

Asyncio model

Traditional threading or processes are not used in the creation of new 'async' entities, so the vulnerabilities associated with failing to initiate new concurrent entities do not apply. Vulnerabilities associated with communication between the 'async' entity and

the initiating entity are addressed in 6.61 Concurrency - data access [CGX] and 6.63 Concurrency – Lock protocol errors [CGM].

The `asyncio.run()` function manages the asyncio event loop. It cannot be called when another `asyncio` event loop is running in the same thread. Its design requires that it be used as the main entry point for `asyncio` programs and only be called once.

If any task in an event loop blocks, it runs the risk of never being resumed if the event loop ends before the block condition expires. Many functions in the Python standard library incur blocking, and therefore are subject to this issue. Therefore, many libraries also exist in non-blocking versions.

Managing multiple `asyncio` events can be error prone. Python provides a debug mode and `logging` module to help identify and catch common issues, as documented in the Python documentation set [5].

Common vulnerabilities of all models

In each of the three forms of concurrency discussed above, there is a risk that some concurrent part of the program will incur an exception. Notification of the main body of the program is uncertain, as described in 6.62 Concurrency -- Premature termination [CGS].

The threat of deadlocks by mutual dependence exists for threads, processes, and analogously for futures. For example:

```
from concurrent.futures import ThreadPoolExecutor
import time

def foo_a():
    time.sleep(1)
    print(b.result())
    return 1

def foo_b():
    print(a.result())
    return 2

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(foo_a)          # waits indefinitely on
b
```

```
b = executor.submit(foo_b)          # waits indefinitely on
a
```

Additional vulnerabilities can arise if a single Python program attempts to use multiple concurrency models, since the different models use different mechanisms for creation, scheduling, communication, and termination.

6.59.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.59.5 for activation of processes or threads or asyncio tasks.
- For any processes and threads that have already been started, ensure that additional starts on that same object are not attempted to avoid exceptions.
- Avoid mixing concurrency models within the same program, or if unavoidable, use with extreme caution.
- Handle all exceptions related to thread creation.
- When using `asyncio`, make all tasks non-blocking and use `asyncio` calls from an event loop.
- Use the debug mode of the Python interpreter to detect concurrency errors.
- To reduce the chance of excessive delays, perform concurrent `asyncio` operations only on non-blocking code.
- When using multiple threads, consider using the `ThreadPoolExecutor` within the `concurrent.futures` module to help maintain and control the number of threads being created.
- For coroutines, ensure that each async call executes operations that relinquish control of the processor when appropriate.

Commented [p61]: Correct reference?

Commented [SJM62R61]: Another way of saying keep all calls non-blocking

6.60 Concurrency – Directed termination [CGT]

6.60.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1:2024 6.60 apply to Python.

Threading model

In Python, a thread may terminate by coming to the end of its executable code or by raising an exception. Python does not have a public API to terminate a thread. This is by design since killing a thread is not recommended due to the unpredictable behaviour that results. There are, however, dangerous workarounds that can terminate Python threads by using calls to the operating system or the `ctypes` foreign

function library. These workaround techniques can lead to deadlock, data corruption, and other unpredictable behaviour as described in ISO/IEC 24772-1:2024 6.60.

The p106referred way to terminate an executing thread is to send it a message, signal or event to terminate itself, and then wait for the termination to occur (using `join()`,).

The parent of a thread can determine if the child has completed either by repeated calls to `is_alive()` or by executing the `join()` statement. The `join()` operation has an optional timeout parameter to reduce the risk of infinite waiting and to provide the possibility for corrective action. The `join()` operation does not return a final result (except `None`), hence joining another thread or process multiple times within the same thread has no effect on the calling entity after the first call which awaited completion of the joined entity.

There are a number of possible errors associated with the joining of threads:

- Failure to join a completed thread can result in logic errors;
- Joining multiple children in an order different than the expected completion of those children can cause extended or indefinite delays;
- Attempting to join the current thread will result in an exception; and
- Any attempts to communicate with another thread after joining that entity can result in significant errors, such as a logic error, an exception, or indefinite delays.

A particular challenge is the scenario of daemon threads. Inside a program, if a thread is created with the flag `daemon = True`, the termination of that thread is disconnected from the termination of the thread that created it. In addition, a `join()` on a daemon thread without a specified timeout will not return.

Multiprocessing model

Since processes are entities of the underlying operating system, terminating other processes is OS-specific. Processes terminate when they complete their program code, but do not notify the creating process; the programmer is responsible to communicate final results or a termination notice before each process terminates.

The preferred way to terminate an executing process is to send it a command to terminate itself, and then wait for the termination to occur using `join()`.

Commented [p63]: Very wrong!!!
Join waits. Join.is-alive checks whether the thread is still running and does not block.

Commented [SJM64R63]: `join()` blocks new threads from *starting* until all currently running threads are completed. Need to discuss.

Here is an example of a graceful shutdown using a simple flag:

```
import threading
import time

def run():
   while True:
      print('thread running')
      global stop_threads
      if stop_threads:
         break

stop_threads = False
t1 = threading.Thread(target = run)
t1.start()
time.sleep(1)
```

... [3]

Commented [SJM65]: 'a given thread'

```
from time import sleep
from threading import Thread

# target function
def task1():
```

... [4]

Commented [SM66R65]: No, "A given thread" includes self.

Commented [p67]: Hmm, see 4 bullet below.
Contradiction? Or does the Rejoining exception not belong there?
Also: should this say "from within the same thread or process".
Presumably, joining from multiple threads is possible?

Commented [SJM68R67]:
From the docs:
https://docs.python.org/3/library/threading.html

"Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

... [5]

Terminating a process in Python is possible but there are scenarios that may leave the system in a vulnerable state:

- Terminating a process that has acquired a lock or semaphore can result in a deadlock condition.
- Executing `terminate()` on a process that is using a pipe or queue may result in lock errors (see 6.63 Lock protocol errors [CGM] or 6.61 Concurrent data access[CGX]).
- Processes that are externally terminated, along with their contained threads, will not execute their `finally` clauses, which can result in logic errors.
- If the terminated process has descendants, then the descendants will be orphaned.

A process can determine if another process has completed either by repeated calls to `multiprocessing.Process.is_alive()` or by calling `multiprocessing.Process.join()`. Calling `join()` with a non-empty timeout together with `is_alive()` permits the calling process to test the progress of the other processes. Calling `join()` with an empty timeout value causes the process to await the completion of the other process.

Asyncio model

Termination of the event loop

When `asyncio` actions are scheduled and the parent is terminated, then the event loop is terminated with a runtime error possibly before some futures are delivered and program termination completes. To achieve controlled termination externally to the event loop, Python recommends terminating the event loop owner with an exception, catch the exception, and send each `asyncio` event a `stop()` or a `run_until_complete()` directive to finish processing already-scheduled events and then cease processing. Once the event loop has completed it can be `close()`'d (after collecting results).

The following example shows another way to terminate an event loop that is interrupted by an exception. In general, such an exception would cause the concurrent iterations to be in an abnormal state. The associated `finally` clause cleans them up and terminates them.

```
Try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

An event loop can also await the completion of a selected set of tasks.

Termination of `asyncio` tasks

To direct the termination of an `asyncio` task, one can set a shared variable that will direct `asyncio` task to terminate itself. The `asyncio` task can:

- Fail to detect the termination request;
- Detect and obey the termination request; or
- Detect and ignore the termination request.

In all cases, the vulnerabilities documented in ISO/IEC 24772-1:2024 6.60 apply to `asyncio` tasks.

Another mechanism is to asynchronously raise the `CancelledError` exception in an asyncio task via the `cancel` method in the `asyncio.Task` class (see example below). If the exception is caught, the recipient task may:

- Complete;
- Report the error condition and complete; or
- Take alternative action and continue processing.

```
import asyncio

async def foo():
    try:
        for i in range (1, 10):
            print("Count...%d" %i)
            await asyncio.sleep(1)
    except asyncio.CancelledError as e:
        print("Stopping foo")
    finally:
        print("foo stopped")

async def main():
    t1 = asyncio.create_task(foo())
    await asyncio.sleep(5)
    t1.cancel() # Cancel count after 5 seconds
    await t1
    print("Hello world")
```

```
if __name__ == '__main__':
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    asyncio.run(main())

OUTPUT:
Count...1
Count...2
Count...3
Count...4
Count...5
Stopping foo
foo stopped
Hello world
```

If the exception is ignored, the recipient task is not permitted to continue executing; it is transferred to its `finally` portion. Vulnerabilities associated with unhandled exceptions are addressed in 6.36 Ignored error status and unhandled exceptions [OYB].

In any of the above cases, the vulnerabilities documented in ISO/IEC 24772-1:2024 6.60 apply to Python `asyncio` tasks.

Common vulnerabilities of all models

The termination of any concurrent activity can consume significant time and resources, e.g. because of finalization. Thus, there is a risk of timing errors for the remaining concurrent entities.

6.60.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.60.5.
- Avoid external termination of concurrent entities except as an extreme measure, such as the termination of the program.
- Use inter-thread or inter-process communication mechanisms to instruct another thread or process to terminate itself.
- Ensure that all shared resources locked by the thread or process are released upon termination, for example, in an exception handler and/or in a finally block.
- Design the code to be fail-safe in the presence of terminating processes, threads or tasks.

- Forbid calls to `join()` on a daemon thread.

6.61 Concurrent data access [CGX]

6.61.1 Applicability to language

The vulnerabilities as documented in ISO/IEC 24772-1:2024 6.61 apply to Python. The traditional accesses to shared data, and the locking and unlocking of locks that protect shared data are as described in ISO/IEC 24772-1:2024 6.61.

Threading model

Threads and events can share memory, and care is required to coordinate the update and consumption of values in such memory. This is not restricted to global data since nesting of threads will effectively make all variables of enclosing threads shared.

Commented [p69]: Add "values in" (otherwise this sounds like heap mgmt.)

Commented [SJM70R69]: Discuss

Some Python interpreters use a GIL which ensures that only a single bytecode is executed at a time. This guarantees that single instruction accesses to primitive data objects are serialized but does not guarantee serialization of data access between threads or asyncio tasks in general.

When using multiple threads, if certain events need to occur sequentially, putting these events into the same thread guarantees sequential access, reduces the need for locks and minimizes the chance for data corruption and race conditions.

When `global` variables are needed to communicate between functions within a single thread in a multithreaded application, visibility of the data to other threads (and the possibility of data corruption and race conditions) can be avoided by using the `threading.local()` function. This creates a local copy of the `global` variable in each thread that executes that call. Threads that do not create a local copy see (and can update) the `global` variable. Confusion can result if some threads maintain a local copy and others do not.

All other shared access to variables requires that the data be locked before access and unlocked after (see 6.63 Lock protocol errors [CGM]).

Multiprocessing model

Python processes do not share memory and therefore are not subject to data access errors between the processes, however, access errors can occur for objects such as those provided by `multiprocessing.sharedctypes` or maintained by the

operating system and shared by processes, such as files. For such objects, the vulnerabilities exist.

Interprocess communication mechanisms such as pipes can exhibit concurrency control errors (see 6.63 Lock protocol errors [CGM]). Note that the use of pipes or queues to move significantly large amounts of data can reduce complexity related to global locks at the expense of performance, which can cause the application to run too slowly and/or miss deadlines.

Pipes and queues are designed such that one process writes to a pipe or queue and a second process reads from it. If one of the processes contains threads, and multiple threads attempt to access the same `pipe` or `queue`, then there is a risk of data corruption since the order of access cannot be guaranteed. Indeed, the use of more than one concurrency model in the same application makes the application susceptible to uncoordinated data accesses.

Asyncio model

A fundamental principle in writing `asyncio` tasks is that each iteration of a task (from the point where its data is ready for processing and where it suspends for the next iteration) is atomic with respect to the other tasks. It is a fundamental error to split calculations or shared data access between iterations of the same task since other tasks can access or change the data between iterations.

6.61.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2024 6.61.5.
- Avoid using `global` variables and consider using the `queue.Queue()`, `threading.queue()()`, `asyncio.queue()()` or `multiprocessing.Queue()()` functions to exchange data between threads or processes respectively.
- If data accesses need to be serialized, ensure that they reside in the same thread, or provide explicit synchronization among the threads or processes for the data accesses.
- For threads:
    - When using multiple threads, verify that all shared data is protected by locks or similar mechanisms.
    - If shared variables must be used in multithreaded applications, use model checking or equivalent methodologies to prove the absence of race conditions.

- Consider using `threading_local()` within each thread in multithreaded code, to create a local copy of each `global` variable that is used as a read-only variable.

- For asyncio:
  - When multiple `asyncio` tasks access data shared among tasks, always complete such access in each task prior to awaiting any event.
  - When multiple `asyncio` tasks access complex data shared among tasks which may require multiple iterations to fully update, retain any partial data local to the task and perform the update only when all data is present.

6.62 Concurrency – Premature termination [CGS]

6.62.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1:2024 6.62 applies to Python. Premature termination of any concurrent part of the program exposes all other portions of the program to the risk of logic errors, regardless of which concurrency model is used in the program. Python provides syntax to detect and diagnose many common premature termination scenarios that will let the program recover and continue, as discussed below.

Threading model

The termination of the main thread awaits the termination of all non-daemon children; it then terminates the daemon children and stops.

Exceptions in a thread at any level can be caught by a `try` clause at the outermost level of that thread; and `finally` clauses will be executed in the presence or absence of exception handling. Exceptions unhandled by a thread cause the invocation of the `thread.exceptHook()` method which can be programmed by the user. The default implementation of `thread.exceptHook()` causes silent termination of the thread.

All these mechanisms provide the opportunity to implement the necessary communication between threads about their termination state.

Any `join()` with the terminated thread is still possible but will not distinguish between normal and exceptional termination. Furthermore, predefined routines such

as `threading.is_alive()`, `threading.active_count()`, and `threading.enumerate()` permit querying the state of other threads.

If termination occurs when a thread is accessing a pipe, then the pipe may become corrupted and further accesses can result in an exception or in undefined behaviour. If termination occurs when a thread is accessing a queue, then the queue may remain locked indefinitely and subsequent accesses can result in deadlock (see 6.63 Lock protocol errors). When using `ThreadPool` objects, it is important to properly manage the resources with a context manager or by calling `close()` and `terminate()` explicitly to prevent deadlock during finalization. Relying on Python's garbage collector to destroy the pool will not guarantee that the finalizer of the pool will be called.

Commented [p71]: Duplicate this to unspecified behavior. Strange, though, to read this specifically for Threadpools. What about other data structures: Is it the case that finalizers for anything might not be called by GC in general?

Commented [SJM72R71]: Also applicable to processes

To prevent premature termination of the child threads, the parent must `join()` each non-daemonic child to wait for them to terminate before proceeding. It is important to prevent Python processes or threads from waiting on daemon processes or threads since the daemons never complete until the program exits.

If a child thread has put items in a queue and it has not used `JoinableQueue.cancel_join_thread`, then that thread will not terminate until all buffered items have been flushed from the queue to the underlying pipe, and future attempts to join that thread may result in a deadlock unless all items in the queue have been consumed.

Multiprocessing model

If the execution of a process incurs an exception and terminates prematurely, then any communicating processes can fail to receive expected results and can suffer from protocol errors, or themselves can wait indefinitely. OS calls to query the state of other processes are available, hence periodic checking whether the other processes are still executable can be used.

Exceptions that occur within a task can notify the parent by using a `try-except` block within the task as shown below:

```
from time import sleep
from multiprocessing.pool import Pool

def task():
    sleep(1)
    # Handle the exception in the task
    try:
        raise Exception()
    except Exception:
        return 'An ERROR occured in task'
```

```
    return 'Task completed successfully.' # unreachable code

if __name__ == '__main__':
    # Create a pool of processes
    with Pool() as pool:
        result = pool.apply_async(task)
        value = result.get()
        print(value)

OUTPUT:
An ERROR occurred in task
```

Similarly, exceptions can also be handled within the parent by using a `try-except` block as shown below:

```
from time import sleep
from multiprocessing.pool import Pool

def task():
    sleep(1)
    raise Exception()
    return 'Task completed successfully.' # unreachable code

if __name__ == '__main__':
    with Pool() as pool:
        result = pool.apply_async(task)
        # Handle task in parent
        try:
            value = result.get()
            print(value)
        except Exception:
            print('An ERROR occurred in task')

OUTPUT:
An ERROR occurred in task
```

Exception handling across process boundaries can also be accomplished by using global objects or the `multiprocessing.Event` flag to communicate between processes.

If an exception occurs in `main()`, child processes can continue to run and should be handled accordingly, such as by catching the exception, terminating and cleaning up all child processes and structures that are the responsibility of this process. If termination occurs when a process is accessing a pipe, then the pipe can become corrupted and further accesses can result in an exception or in undefined behaviour. If termination occurs when a process is accessing a queue, then the queue is likely to remain locked indefinitely and subsequent accesses can result in deadlock (see 6.63 Protocol lock errors [CGM]).

When using `multiprocessing.pool` objects, it is important to properly manage the resources with a context manager or by calling `close()` and `terminate()` manually to prevent deadlock during finalization. Processes that terminate cannot be restarted. Relying on Python's garbage collector to destroy the pool will not guarantee that the finalizer of the pool will be called.

Asyncio model

Premature termination occurs as follows:

- When the primary task terminates due to an exception or unprogrammed event;
- When a dependent task raises an exception or terminates abnormally.

For the first scenario, all dependent tasks will be terminated when the main task terminates (see 6.36 Ignored error status or unhandled exception [OYB]).

For the second scenario, the premature termination of dependent coroutines will almost always affect the execution of `main()` and other coroutines. If all tasks are not cooperatively terminating, then it is unlikely that the program will execute correctly.

The following methods can be helpful in handling `asyncio` exceptions:

- `get_name()` – Returns the name of the Task
- `exception()` – Returns the exception of the Task, or returns `None` if there are no exceptions.
- `result()` – Returns the result of the Task coroutine or `None` if the coroutine does not have a return. If the task has been cancelled, a `CancelledError` exception is raised. If the result is not completed, an `InvalidStateError` is raised. All exceptions are re-raised so that they can propagate back to the caller for handling.

When `main()` calls two or more coroutines, precautions need to be taken since an exception in any coroutine gets sent to the scheduler and then handled by `main()` only after the `return_when` condition is satisfied. If `main()` does not recognize an

exception from a subordinate coroutine, it will not get handled and will remain in the event loop for the remainder of the program. The following example uses the above methods to help ensure that `main()` gets notified and all tasks are removed from the event loop prior to program termination.

```
import asyncio

async def coro1():
    raise RuntimeError("ERROR in coro1")
    return ("coro1 completed")  # Unreachable code

async def coro2():
    await asyncio.sleep(1)
    return ("coro2 completed")

async def main():
    # Create tasks
    t1 = asyncio.create_task(coro1(), name='task1')
    t2 = asyncio.create_task(coro2(), name='task2')
    tasks = [t1, t2]

    # Run both tasks concurrently and block until the
condition
    # specified by return_when (ALL_COMPLETED in this case)
met.
    done, pending = await asyncio.wait(\
                           tasks, return_when =
asyncio.ALL_COMPLETED)
    # Handle all 'done' tasks
    for task in done:
        # Get name of the task that was assigned during
creation.
        task_name = task.get_name()
        print(task_name, "is done")
        # Obtain exception object raised by coroutine
        exception = task.exception()
        # Print the task name associated with any exceptions
        if isinstance(exception, Exception):
            print(task_name, "threw following exception:",
exception)
```

```
        # Test for errors
        try:
            # Returns result of coroutine & re-throws
exceptions
            # that may have occurred so that they can be
handles.
            result = task.result()
            print(task_name, "returned:", result)
        # Print errors that may occur
        except RuntimeError as err:
            print("RuntimeError:", err)
    # Handle 'pending' tasks
    for task in pending:
        task.cancel()

asyncio.run(main())
```

The above example shows that even though both tasks are reported to be done, the exception only gets passed to `main()` by calling `task.result()`. The example runs successfully and produces the following output:

```
task2 is done
task2 returned: coro2 completed
task1 is done
task1 threw the following exception: ERROR in coro1
RuntimeError: ERROR in coro1
```

6.62.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.62.5.
- Protect data that would be vulnerable to premature termination, such as by using locks or protected regions, or by retaining the last consistent version of the data (checkpoints).
- Enable event logging and record all events prior to termination so that full traceability is preserved.
- For threads:
    - Handle exceptions; free locks; and clean up nested threads and shared data before termination.

- Use the `try` or `finally` clauses in thread methods and consider notifying a higher-level construct of the termination so that any corrective action if needed can be taken.
- Consider using one or more of the
  `Thread.is_alive()`,
  `Thread.active_count()`,
  `Thread.enumerate()`
  methods in `threading` to determine if child threads' execution states are as expected.
- Finalize thread pools before destroying them.

- For multiprocessing:
  - Handle exceptions; free locks; and clean up any processes that are the responsibility of this process.
  - Use the `try` or `finally` clauses in process methods and consider notifying a higher-level construct of the termination so that any corrective action if needed can be taken.
  - Consider using one or more of the
    `Process.is_alive()`,
    methods in `multiprocessing` to determine if child process' execution states are as expected.
  - Finalize process pools before destroying them.

- For asyncio:
  - Ensure consistent termination behaviour of all coroutines

6.63 Lock protocol errors [CGM]

6.63.1 Applicability to language

The vulnerabilities as documented in ISO/IEC 24772-1:2024 6.63 apply to Python.

Python provides locks and semaphores that are intended to protect critical sections managing shared data. All calls to `lock.acquire()` with default parameters guarantee that the calling concurrent unit (thread, process, or coroutine) will not

Commented [SM73]: SM - check if there are any other calls needed here.

Commented [SJM74R73]: NO other calls needed
https://docs.python.org/3/library/multiprocessing.html

`is_alive()`
Return whether the process is alive.
Roughly, a process object is alive from the moment the `start()` method returns until the child process terminates.

continue until the lock is available. Python also provides event objects that permit programmed-specific notification between two concurrent units, as well as barriers and condition objects that permit the release of groups of concurrent units upon a single condition becoming true. However, there are vulnerabilities associated with Python's synchronization mechanisms:

- If a concurrent unit is killed in between `lock.acquire()` and `lock.release()`, every other concurrent unit unconditionally waiting on that lock will be deadlocked.
- Locations where locks are needed can be missed, unless shared resources are accessed exclusively by dedicated functions that act like a traditional monitor.
- The use of locks does not guarantee consistency of shared resources unless all relevant concurrent units check for the locks.
- Every critical section that starts with a `lock.acquire()` must be matched with a `lock.release()`, or the program, or some concurrent units, will deadlock.
- For calls of `lock.acquire(..)` that are parameterized with a time-limit or with the requirement for immediate locking, the omission of checking the result of `lock.acquire(..)` will allow the caller to proceed without acquiring a lock.

Threading model

Multiple threads can have shared data, as well as other shared resources. All of the vulnerabilities documented in ISO/IEC 24772-1:2024 6.63 apply. In particular, access by multiple threads to the same pipe or queue exhibits these vulnerabilities.

To avoid the vulnerabilities, concurrent access to such data or resources must be synchronized. The following example shows a simple scenario where synchronization is required.

```
database_value=0
lock=threading.Lock()

def update(x):…
     #Takes a finite amount of time and updates x

def increase():
     global database_value
     global lock
     lock.acquire()
```

```
        local_copy = database_value
        update(local_copy)
        database value = local_copy
        lock.release() # don't forget this else deadlock
```

A better alternative is to use a context manager since it acquires and releases the `lock` automatically.

```
def increase():
    global database_value
    global lock
    with lock: # The context manager.
        local_copy = database_value
        update(local_copy)
        database_value = local_copy

if __name__ == "__main__":
    print('start value', database_value)
    thread1 = Thread(target=increase)
    thread2 = Thread(target=increase)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    print('end value', database_value)
    print('end main')
```

Threads that have been created typically need to return a result. This is often accomplished via the `join()` method. There are a number of possible errors associated with the joining of threads:

- Joining multiple child threads in an order different than the expected completion of those children can cause extended or indefinite delays.
- Attempting to `join()` the current thread will result in an exception.
- Using `join()` on a daemon thread will result in a deadlock condition.
- Attempting to `join()` a thread before starting it will result in a runtime error.

Commented [p75]: Not a rejoin exception? As stated elsewhere

Commented [SJM76R75]: https://docs.python.org/3/library/threading.html#threading.Thread.join

RuntimeError: cannot join current thread

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock.

Multiprocessing model

Multiple processes in Python do not have shared data, thus no synchronization is required to access such data. However, other resources such as OS-level variables or files, can be accessed by multiple processes and require synchronization. In order to communicate data or state between processes, Python provides API's for pipes, queues, and files. Some mechanisms (i.e. queues) are designed to be usable by multiple processes by encapsulating the interface to each in multiprocess-safe calls. For pipes and files, Python does not provide automatic synchronization between multiple readers or writers of the pipe or file, and thus explicit synchronizations is the responsibility of the programmer. Process locks or process semaphores can be used to guarantee exclusivity.

The issues related to multiple threads attempting to access the same interprocess communication abstraction are discussed above under "Threading model".

Processes that have been created usually need to return a result. This is accomplished via the `join()` method (see 6.61 Concurrency – data access [CGX]). There are several possible errors associated with the joining of threads or processes:

- Joining multiple child processes in an order different than the expected completion of those children can cause extended or indefinite delays.
- Attempting to `join()` the current process will result in an exception.
- Using `join()` on a daemon process will result in an exception.
- Attempting to `join()` a process before starting it will result in a runtime error.

Commented [p77]: ditto

Commented [SJM78R77]: Rejoin

RuntimeError: cannot join current thread

Asyncio model

Although Python provides mechanisms for `asyncio` tasks to control access to data or resources shared between them, such usage can result in serious errors and vulnerabilities. The coroutine model of programming associates a single `asyncio` task with a single IO event and communicates results directly back to the initiator of the task. The scheduler takes responsibility for the scheduling of multiple tasks and ensures that they cannot access shared resources concurrently.

Nevertheless, coroutines can be programmed to access state or resources that are not coroutine-safe. For example, some programming models have coroutines that interact with each other or with multiple IO events before relinquishing control to the event loop. In such cases, it is necessary to identify critical regions where the order of access by different coroutines matter, and locks of such regions are necessary.

The `asyncio` module provides the `asyncio.Lock` class to protect these critical sections, but these sections are not thread-safe or process-safe, hence cannot be safely shared by any other thread or process or their respective `asyncio` tasks. The same

instance of the `asyncio.Lock` class must be used by all coroutines that access a shared resource so that race conditions can be avoided.

As communicating coroutines execute within a single thread, calls on blocking functions (other than `await`) will block the thread (and all other coroutines of the thread).

6.63.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2024 6.63.5.
- Verify that all sections of code that have critical sections check the related lock prior to entering the critical section, including API calls known to be unsynchronized, and release the acquired lock at the end of the section.
- Avoid intermixing concurrency models within the same Python program, including programs that are replicated across multiple processes to gain access to multicore hardware.

Threading model

- If global variables are used in multi-threaded code, consider using locks or semaphores in a module that contains all operations on them so that all accesses are serialized.
- Avoid explicit coding of locks by encapsulating all related global data in monitor-like structures (as published in the literature).
- For threads, use `join()` as the final interaction with other thread(s) to ensure that the calling thread is blocked until all joined threads have either terminated normally, thrown an exception, or timed out (if implemented).
- Ensure that `join()` is not used on a thread before it is started since this will throw an exception.
- When using `Pipe()` in conjunction with threads, restrict the writing of a single pipe to a single thread, and similarly for reading.

Multiprocessing Model

- Ensure that `join()` is not used on a process before it is started since this will throw an exception.
- When using `Pipe()` in conjunction with processes or threads inside multiple processes, restrict the writing of a single pipe to a single thread per process, and similarly for reading.
- If exclusive access to any resource shared among multiple processes is needed, ensure the exclusivity by synchronization mechanisms provided by the multiprocessing module.

Asyncio model

- Forbid `await` or `sleep` within critical sections.
- Prefer a programming model such that the event loop is responsible for the distribution and post-processing of all data collected by `asyncio` tasks. Such post-processing can be delegated to other tasks.
- Forbid `asyncio` coroutines from invoking any blocking construct except the `await` statement.

6.64 Reliance on external format string [SHL]

6.64.1 Applicability to language

The vulnerabilities as documented in ISO/IEC 24772-1:2024 6.64 apply to Python. Externally controllable strings can result in unexpected behaviour such as buffer overruns, exposure of private data, and other malicious exploits. Python strings share most of the potential security vulnerabilities described in ISO/IEC 24772-1:2024 6.64.

6.64.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.64.3.
- Implement checks to limit the size of input strings so that they do not exceed the expected length.
- Review the Python format string specifiers and forbid vulnerable formats provided by the user.

6.65 Modifying constants [UJO]

6.65.1 Applicability to language

This vulnerability as documented in ISO/IEC 24772-1:2024 6.65 minimally applies to Python because Python has only a small number of constants.

Python does not allow the declaration of constants. However, Python has six constants declared as part of the language. The list is:

```
False
True
None
NotImplemented
Ellipsis (same as the ellipsis literal "...")
__debug__
```

Note that per the Python language documentation: "Changed in version 3.9: Evaluating `NotImplemented` in a boolean context is deprecated. While it currently evaluates as `True`, it will emit a `DeprecationWarning`. It will raise a `TypeError` in a future version of Python."

Early versions of Python would allow these constants to be given a new value. Since Python version 3.0, the first three, `False`, `True` and `None`, have been declared as keywords in addition to being a constant so their values may no longer be changed. The remaining three, `NotImplemented`, `Ellipsis` and `__debug__`, can be assigned new values without raising a `SyntaxError` making them modifiable constants.

6.65.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Apply the avoidance mechanisms provided by ISO/IEC 24772-1:2024 6.65.3.
- Forbid assigning new values to `NotImplemented`, `Ellipsis` or `__debug__`.

7. Language specific vulnerabilities for Python

7.1 General

This clause documents vulnerabilities specific to Python that are not yet addressed in ISO/IEC 24772-1.

7.2 Lack of Explicit Declarations

7.2.1 Description of application vulnerability

As explained in 5.1.4, an assignment to a not yet existing variable is legal and creates the variable and its object at that location. This capability also extends to the data members of a class, thereby extending that class. Moreover, reassigning an existing label to a different object binds the label to the new object regardless of the type of the previous object. Hence, any arbitrary assignment to a variable is legal.

7.2.2 Cross reference

7.2.3 Mechanism of failure

A mistyped label name as the target of an assignment simply introduces a new label. For example, upon execution of

```
CountTheNumberOfObjects = 0
   # and later on …
CountTheNumberofObjects = CountTheNumberOfObjects + 1
# Two different variables, capital vs. lowercase "O" in
"Of"!!!
```

Most programmers will miss small and unintentional differences in the names and be highly surprised by the fact that `CountTheNumberOfObjects` will retain its initialized value, usually `0`.

Thus any unintentional mistyping of identifiers on the left hand side of an assignment is required by the language to go unnoticed. However, reading the value of an unknown variable will result in runtime error `NameError`.

7.2.4 Avoiding the vulnerability or mitigating its effects

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Use consistent naming conventions, such as if using camel case, the first letter of all words should always be capitalized.
- Be cognizant of the number of significant characters in variables and consider staying below the limit for the number of significant characters.

7.3 Code representation differs between compiler view and reader view

7.3.1 Description of application vulnerability

The ISO/IEC 10646:2020 character set, which Python supports, includes characters that can effectively hide adjoining text. Such characters set text display direction left-to-right or right-to-left but are invisible unless the editor or display program is instructed to mnemonically display them. If left-to-right is the current default direction and a right-to-left character (`RLI`) is used, subsequent text will visually replace the text preceding the `RLI` character.

The following example, taken from [1], shows code with the invisible characters denoted visibly by `+LRI, +PDI, +RLO`, where these denotations stand for the zero-space Unicode control characters:

```
<LRI> Left-to-Right Isolate
<PDI>  Pop Directional Isolate
<RLO>  Right-to-Left Overwrite
```

Due to the direction-changing characters, the following code

```
alvl = 'user'
if alvl != 'none+RLO+LRI': #Check if admin+PDI+LRI' and alvl!= 'user’
    print('You are an admin.')
```

will be displayed to the human reader in some editors as:

```
alvl = 'user'
if alvl != 'none’ and alvl!= 'user' #Check if admin
    print('You are an admin.')
```

However, this code will always print "`You are an admin`", as the apparent second condition is really part of a comment in the original code.

Python only permits the use of direction-changing control characters in comments and strings. Nevertheless, malicious use can change string or comment into executable code, as shown above and also below using RLI in a string.

```
'''Subtract funds from account then  RLI      ''' ; return
'''LRI'''
```

This line can display as, depending on the text editor used;

```
'''Subtract funds from bank account then return;'''
```

but executes as

```
; return
```

A similar situation arises from the use of the carriage return <CR> and line feed <LF> characters, depending upon the environment where the code is executed.

Example

```
Blow_Up(); <CR> BeReallyNice()
```

The lack of a <LF> can cause the code (e.g in UNIX-based systems) to be displayed as

```
BeReallyNice()
```

while the code executes as

```
Blow_Up(); BeReallyNice()
```

because some environments will overwrite the physical line if the <LF> is not included.

7.3.4 Avoiding the vulnerability or mitigating its effect

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Carefully manage and thoroughly review the use of any characters that can in any way hide the functionality and representation of Python code.
- Avoid reliance on simple visual inspection of code; instead use tools to reveal dangerous control characters.
- Always use static analysis tools that identify all occurrences of hidden characters within a program.

- Use only editors that are capable of revealing the hidden Unicode (zero-space) control characters and ensure that the editor setting is enabled.
- Refrain from copying and pasting code from untrusted sources unless the code is thoroughly checked as described above.

7.4 Time representation and Usage in Python

7.4.1 Description of application vulnerability

The vulnerability described in ISO/IEC 24772-1:2024 7.33 applies to Python.

In addition to the issues documented in ISO/IEC 24772-1:2024 7.33, Python has naïve datetime objects that do not specify a time zone, and thus do not contain enough information to unambiguously provide locale relative to other date and time objects. Such objects can be passed to functions that expect datetime objects of a different locale and thus generate erroneous results.

Aware datetime objects contain timezone information which mitigates the vulnerability.

7.4.2 Cross reference

7.4.3 Mechanism of failure

Python 3 allows naive datetime objects to be used with operations that assume the existence of a timezone in the object, or operations that expect naïve datetime objects and receive a datetime object from a different timezone. In either case, an incorrect datetime value will arise. Examples are aware datetime objects created with the TZ for UTC or naïve datetime object created in the UTC timezone. Such objects when processed by an operation in the EDT timezone that expects naïve datetime objects will be 5 hours off the local time.

Methods such as `utcnow()` and `utcfromtimestamp()` are potentially dangerous since they create a naive datetime and do not throw an error when used in operations expecting non-UTC time objects. These functions are being deprecated by the Python designers for future releases.

When anything other than aware datetime objects and functions are used, time-related values can be calculated incorrectly and routines based upon their correct calculation can fail with arbitrary consequences.

7.4.4 Avoiding the vulnerability or mitigating its effects

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Follow the advice of ISO/IEC 24772-1 7.33.4;
- Avoid the use of naïve datetime objects and functions;
- Place appropriate assertions upon any datetime objects received or processed;
- Avoid deprecated functions.

Bibliography

[1] Anderson, R. & Boucher, N. Trojan Source: Invisible Vulnerabilities, https://trojansource.codes/trojan-source.pdf

[2] Einarsson, B., Accuracy and Reliability in Scientific Computing, SIAM, July 2005 http://www.nsc.liu.se/wg25/book

[3] Embedding Python in Another Application, http://docs.python.org/3/extending/embedding.html

[4] ISO/IEC 60559:2020, Information technology Microprocessor Systems Floating-Point arithmetic

[5] Logging facility for Python, https://docs.python.org/3/library/logging.html

[6] Lutz, M., Learning Python, 5th Edition, Sebastopol, CA: O'Reilly Media, Inc., 2013

[7] Lutz, M., Programming Python, 4th Edition, Sebastopol, CA: O'Reilly Media, Inc., 2010

[8] MITRE Corporation, Common Weakness Enumeration, http://cwe.mitre.org

[9] Packaging binary extensions, https://packaging.python.org/en/latest/guides/packaging-binary-extensions/

[10] PEP 8 - Style Guide for Python Code, http://www.python.org/dev/peps/pep-0008

[11] PEP 551 - Security transparency in the Python runtime, https://www.python.org/dev/peps/pep-0551 (Status: Withdrawn)

[12] PEP 578 – Python Runtime Audit Hooks, https://peps.python.org/pep-0578/

[13] Martelli, A. Python in a Nutshell, Sebastopol, CA: O'Reilly Media, Inc., 2006.

[14] Python/C API Reference Manual, http://docs.python.org/py3k/c-api

[15] The Python Language Reference, https://docs.python.org/3/reference

[16] The Python Standard Library, https://docs.python.org/3/library

[17] Sebesta, Robert W., Concepts of Programming Languages, 11th edition, ISBN-13: 978-0-133-94302-3, ISBN-10: 0-133-94302-X, Pearson Education, Boston, MA, 2015

Commented [p79]: For Sean to fix;

Commented [SJM80R79]: Done?

Commented [SM81]: Done.

[18] Sun Microsystems, Inc. , What Every Computer Scientist Should Know About Floating-Point Arithmetic, Part No: 800-7895-10 Revision A, June 1992, https://docs.oracle.com/cd/E19957-01/800-7895/800-7895.pdf

Index

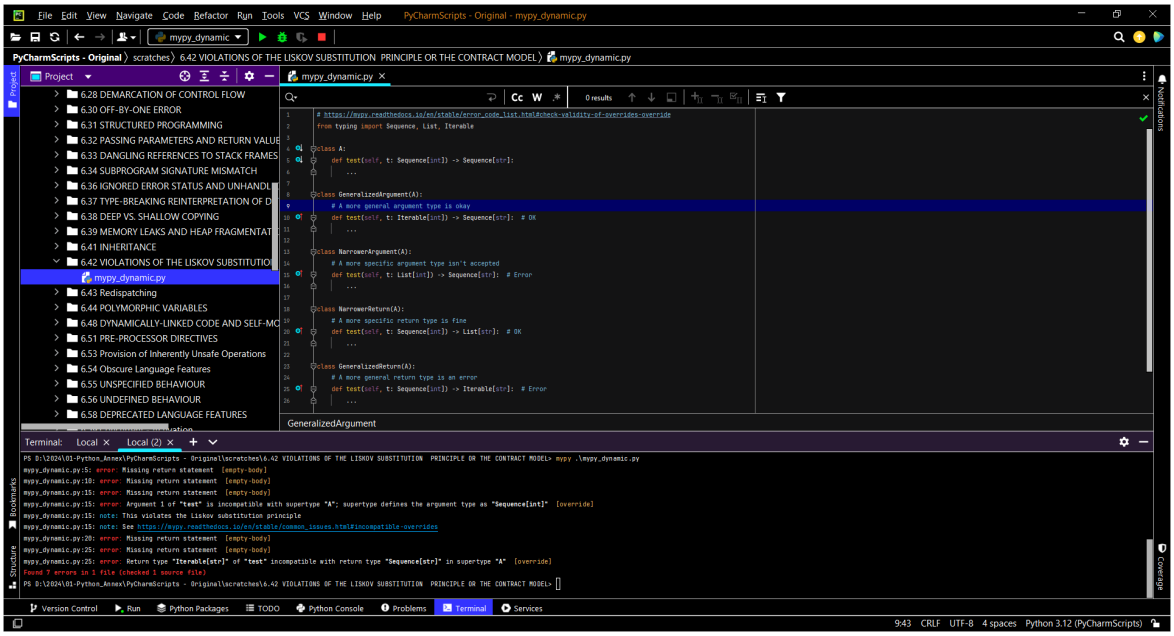
Commented [SM82]: Ensure that all font is normal font here.

Page 3: [1] Deleted **Stephen Michell 6/26/24 2:31:00 PM**

Page 82: [2] Commented [SJM48R47] **McDonagh, Sean 8/8/24 12:51:00 PM**

The following screenshot illustrates mypy in use and how it finds Liskov violations in the example found in:

https://mypy.readthedocs.io/en/stable/error_code_list.html#check-validity-of-overrides-override



Page 106: [3] Commented [SJM64R63] **McDonagh, Sean 7/17/24 12:53:00 PM**

`join()` blocks new threads from *starting* until all currently running threads are completed. Need to discuss.

Here is an example of a graceful shutdown using a simple flag:

```
import threading

import time
```

```
def run():
    while True:
        print('thread running')
        global stop_threads
        if stop_threads:
            break

stop_threads = False
t1 = threading.Thread(target = run)
t1.start()
time.sleep(1)
stop_threads = True
t1.join()
print('thread killed')
```

Page 106: [4] Commented [SJM65] McDonagh, Sean 9/11/24 12:08:00 PM

'a given thread'

```
from time import sleep
from threading import Thread

# target function
def task1():
    sleep(2)
    print('thread1: Done')

def task2():
    sleep(1)
    print('thread2: Done')

thread1 = Thread(target=task1)
thread2 = Thread(target=task2)
thread1.start()
thread2.start()
print('Main: Waiting for threads to complete...')
thread1.join()
thread2.join()
thread2.join() # redundant join() on a thread are permitted but meaningless
# thread2.start() # RuntimeError: threads can only be started once
print('Main thread: Done')
```

Page 106: [5] Commented [SJM68R67] McDonagh, Sean 7/30/24 9:42:00 PM

From the docs:

https://docs.python.org/3/library/threading.html

"Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be joined many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raise the same exception."

The previous example in the comments shows at least one way that a thread can communicate with another thread to safely shut it down. I suspect this statement is attempting to state that you cannot join a running thread multiple times. If this is the intent, it needs reworded.