30.1. rexec — Restricted execution framework¶

Deprecated since version 2.6: The `rexec` module has been removed in Python 3.0.

Changed in version 2.3: Disabled module.

Warning

The documentation has been left in place to help in reading old code that uses the module.

This module contains the `RExec` class, which supports `r_eval()`, `r_execfile()`, `r_exec()`, and `r_import()` methods, which are restricted versions of the standard Python functions `eval()`, `execfile()` and the `exec` and `import` statements. Code executed in this restricted environment will only have access to modules and functions that are deemed safe; you can subclass `RExec` to add or remove capabilities as desired.

Warning

While the `rexec` module is designed to perform as described below, it does have a few known vulnerabilities which could be exploited by carefully written code. Thus it should not be relied upon in situations requiring “production ready” security. In such situations, execution via sub-processes or very careful “cleansing” of both code and data to be processed may be necessary. Alternatively, help in patching known `rexec` vulnerabilities would be welcomed.

Note

The `RExec` class can prevent code from performing unsafe operations like reading or writing disk files, or using TCP/IP sockets. However, it does not protect against code using extremely large amounts of memory or processor time.

class `rexec.RExec`([*hooks*[, *verbose*]])¶

Returns an instance of the `RExec` class.

hooks is an instance of the `RHooks` class or a subclass of it. If it is omitted or `None`, the default `RHooks` class is instantiated. Whenever the `rexec` module searches for a module (even a built-in one) or reads a module’s code, it doesn’t actually go out to the file system itself. Rather, it calls methods of an `RHooks` instance that was passed to or created by its constructor. (Actually, the `RExec` object doesn’t make these calls — they are made by a module loader object that’s part of the `RExec` object. This allows another level of flexibility, which can be useful when changing the mechanics of `import` within the restricted environment.)

By providing an alternate `RHooks` object, we can control the file system accesses made to import a module, without changing the actual algorithm that controls the order in which those accesses are made. For instance, we could substitute an `RHooks` object that passes all filesystem requests to a file server elsewhere, via some RPC mechanism such as ILU. Grail’s applet loader uses this to support importing applets from a URL for a directory.

If *verbose* is true, additional debugging output may be sent to standard output.

It is important to be aware that code running in a restricted environment can still call the `sys.exit()` function. To disallow restricted code from exiting the interpreter, always protect calls that cause restricted code to run with a `try`/`except` statement that catches the `SystemExit` exception. Removing the `sys.exit()` function from the restricted environment is not sufficient — the restricted code could still use `raise SystemExit`. Removing `SystemExit` is not a reasonable option; some library code makes use of this and would break were it not available.

See also

Grail Home Page
Grail is a Web browser written entirely in Python. It uses the `rexec` module as a foundation for supporting Python applets, and can be used as an example usage of this module.

30.1.1. RExec Objects¶

`RExec` instances support the following methods:

`RExec.r_eval`(*code*)¶
code must either be a string containing a Python expression, or a compiled code object, which will be evaluated in the restricted environment’s `__main__` module. The value of the expression or code object will be returned.

`RExec.r_exec`(*code*)¶

code must either be a string containing one or more lines of Python code, or a compiled code object, which will be executed in the restricted environment's `__main__` module.

`RExec.r_execfile`(*filename*)¶
Execute the Python code contained in the file *filename* in the restricted environment's `__main__` module.

Methods whose names begin with `s_` are similar to the functions beginning with `r_`, but the code will be granted access to restricted versions of the standard I/O streams `sys.stdin`, `sys.stderr`, and `sys.stdout`.

`RExec.s_eval`(*code*)¶
code must be a string containing a Python expression, which will be evaluated in the restricted environment.

`RExec.s_exec`(*code*)¶
code must be a string containing one or more lines of Python code, which will be executed in the restricted environment.

`RExec.s_execfile`(*code*)¶
Execute the Python code contained in the file *filename* in the restricted environment.

`RExec` objects must also support various methods which will be implicitly called by code executing in the restricted environment. Overriding these methods in a subclass is used to change the policies enforced by a restricted environment.

`RExec.r_import`(*modulename*[, *globals*[, *locals*[, *fromlist*]]])¶
Import the module *modulename*, raising an `ImportError` exception if the module is considered unsafe.

`RExec.r_open`(*filename*[, *mode*[, *bufsize*]])¶
Method called when `open()` is called in the restricted environment. The arguments are identical to those of `open()`, and a file object (or a class instance compatible with file objects) should be returned. `RExec`'s default behaviour is allow opening any file for reading, but forbidding any attempt to write a file. See the example below for an implementation of a less restrictive `r_open()`.

`RExec.r_reload`(*module*)¶
Reload the module object *module*, re-parsing and re-initializing it.

`RExec.r_unload`(*module*)¶
Unload the module object *module* (remove it from the restricted environment's `sys.modules` dictionary).

And their equivalents with access to restricted standard I/O streams:

`RExec.s_import`(*modulename*[, *globals*[, *locals*[, *fromlist*]]])¶
Import the module *modulename*, raising an `ImportError` exception if the module is considered unsafe.

`RExec.s_reload`(*module*)¶
Reload the module object *module*, re-parsing and re-initializing it.

`RExec.s_unload`(*module*)¶
Unload the module object *module*.

30.1.2. Defining restricted environments¶

The `RExec` class has the following class attributes, which are used by the `__init__()` method. Changing them on an existing instance won't have any effect; instead, create a subclass of `RExec` and assign them new values in the class definition. Instances of the new class will then use those new values. All these attributes are tuples of strings.

`RExec.nok_builtin_names`¶
Contains the names of built-in functions which will *not* be available to programs running in the restricted environment. The value for `RExec` is `('open', 'reload', '__import__')`. (This gives the exceptions, because by far the majority of built-in functions are harmless. A subclass that wants to override this variable should probably start with the value from the base class and concatenate additional forbidden functions — when new dangerous built-in functions are added to Python, they will also be added to this module.)

`RExec.ok_builtin_modules`¶
Contains the names of built-in modules which can be safely imported. The value for `RExec` is `('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'select', 'sha', '_sre', 'strop', 'struct', 'time')`. A similar remark about overriding this variable applies — use the value from the base class as a starting point.

`RExec.ok_path`¶
Contains the directories which will be searched when an `import` is performed in the restricted environment. The value for `RExec` is the same as `sys.path` (at the time the module is loaded) for unrestricted code.

`RExec.ok_posix_names`¶
Contains the names of the functions in the `os` module which will be available to programs running in the restricted environment. The value for `RExec` is `('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid')`.

`RExec.ok_sys_names`¶
Contains the names of the functions and variables in the `sys` module which will be available to programs running in the restricted environment. The value for `RExec` is `('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint')`.

`RExec.ok_file_types`¶

Contains the file types from which modules are allowed to be loaded. Each file type is an integer constant defined in the `imp` module. The meaningful values are `PY_SOURCE`, `PY_COMPILED`, and `C_EXTENSION`. The value for `RExec` is `(C_EXTENSION, PY_SOURCE)`. Adding `PY_COMPILED` in subclasses is not recommended; an attacker could exit the restricted execution mode by putting a forged byte-compiled file (`.pyc`) anywhere in your file system, for example by writing it to `/tmp` or uploading it to the `/incoming` directory of your public FTP server.

30.1.3. An example¶

Let us say that we want a slightly more relaxed policy than the standard `RExec` class. For example, if we're willing to allow files in `/tmp` to be written, we can subclass the `RExec` class:

```
class TmpWriterRExec(rexec.RExec):
   def r_open(self, file, mode='r', buf=-1):
       if mode in ('r', 'rb'):
           pass
       elif mode in ('w', 'wb', 'a', 'ab'):
           # check filename : must begin with /tmp/
           if file[:5]!='/tmp/':
               raise IOError("can't write outside /tmp")
           elif (string.find(file, '/../') >= 0 or
                file[:3] == '../' or file[-3:] == '/..'):
               raise IOError("'..' in filename forbidden")
       else: raise IOError("Illegal open() mode")
       return open(file, mode, buf)
```

Notice that the above code will occasionally forbid a perfectly valid filename; for example, code in the restricted environment won't be able to open a file called `/tmp/foo/../bar`. To fix this, the `r_open()` method would have to simplify the filename to `/tmp/bar`, which would require splitting apart the filename and performing various operations on it. In cases where security is at stake, it may be preferable to write simple code which is sometimes overly restrictive, instead of more general code that is also more complex and may harbor a subtle security hole.