Restricted Execution HOWTO

Release 2.0

A.M. Kuchling

April 17, 2002

akuchlin@mems-exchange.org

Abstract

Python provides a restricted execution mode for running untrusted code that will prevent the code from performing dangerous operations. This HOWTO explains how to use restricted execution mode, and how to customize the restricted environment for your application. It aims to provide a gentler introduction than the corresponding section in the Python Library Reference.

This document is available in several formats, including PostScript, PDF, HTML and plain ASCII, from the Python HOWTO page at http://www.python.org/doc/howto/.

Contents

1 Basic use of `RExec`

For some applications, it's desirable to execute chunks of Python code that come from an outside source. The most obvious example is a Web browser such as Grail, which can download and execute applets written in Python.

An obvious danger of downloading and running code from anywhere is that someone might write a malicious applet that appears to be harmless, but silently erases files, makes copies of sensitive data, or gives the applet's author a back door into your system. The solution is to run the code in a restricted environment, where it's prevented from performing any operations that could be used maliciously.

Java does this by using the Java Virtual Machine, which executes Java bytecode. The virtual machine, or VM, has complete control over the running applet, and any dangerous operations must go through the VM in order to be performed. The VM can therefore trap suspicious activity, and stop the applet's execution, if a strict security policy is used, or ask the user if the operation should be permitted, if the policy is somewhat looser.

Python already has a virtual machine that executes Python byte codes, so creating a restricted execution environment simply requires sealing off dangerous built-in functions such as `open()`, and dangerous modules, such as the `socket` module. This can be done by creating new namespaces, removing any dangerous functions, and forcing code to be executed in those namespaces. While a simple idea, in practice it's fairly complicated to implement. Luckily, the required features have been present in Python for a while, and it's already been implemented for you as a standard module.

Code for using a restricted execution environment is in the 'rexec' module. The base class is called `RExec`; in a later section of this HOWTO, we'll show you how to create your own subclasses of `RExec` to customize the functions and modules that are available. Here's the documentation for creating a new `RExec` instance:

`RExec``(`*[hooks], [verbose]* `)`

Returns a `RExec` instance. The *verbose* parameter is a Boolean value, defaulting to false. If true, the `RExec` instance will execute in verbose mode, which will print a debugging message when modules are imported, as if the `-v` option was given to the Python interpreter.

The *hooks* parameter can be an instance of the `RHooks` class, or of some subclass of `RHooks`; a default instance will be used if the parameter is omitted. This is only required when creating particularly exotic restricted environments that import modules in new ways. If you need to use this, you'll have to consult the source code (or Guido) for a complete picture of what's going on.

The `RExec` instance has `r_exec()`, `r_eval()`, and `r_execfile()` functions, which do the same thing as Python's built-in `exec()`, `eval()`, and `execfile()` functions, performing them in the restricted environment. (There are also `s_exec()`, `s_eval()`, and `s_execfile()` methods which replace the restricted environment's standard input, output, and error files with `StringIO` objects that allow you to control the input and capture any output generated.)

Here's a sample usage of a restricted environment. First, the `RExec` instance has to be created.

```
r_env = rexec.RExec()
```

Now, we can execute code and evaluate expressions in the environment:

```
r_env.r_exec('import string')
expr = 'string.upper("This is a test")'
print r_env.r_eval( expr )
```

The first line executes a statement, importing the `string` module. Since it's considered a safe module, the operation succeeds. The second and third lines create a string containing an expression, and evaluates the expression in the restricted environment; it prints out '`THIS IS A TEST`', as you'd expect.

Unsafe operations trigger an exception. For example:

```
r_env.r_exec('import socket')
```

The previous line will cause an `ImportError` exception to be raised, with an associated string value that reads "untrusted dynamic module: _socket". Trying to open a file for writing is also forbidden:

```
r_env.r_exec('file = open("/tmp/a.out", "w")')
```

This will raise an `IOError` exception, with an assocated string value that reads "can't open files for writing in restricted mode". The restricted code can catch the exception in a `try...except` block and continue running; this is useful for writing code which works in both restricted and unrestricted mode. Opening files for reading will work, however.

Exactly what restrictions does the base `RExec` impose? It limits the modules that can be imported to the following safe list:

```
audioop, array, binascii, cmath, errno, imageop,
marshal, math, md5, operator, parser, regex,
pcre, rotor, select, strop, struct, time
```

In general, these are modules that can't affect anything outside of the executing code; they allow various forms of computation, but don't allow operations that change the filesystem or use network connections to other machines. (The `pcre` module may be unfamiliar. It's an internal module used by the `re` module, so restricted code can still use the `re` to perform regular expression matches.)

It also restricts the variables and functions that are available from the `sys` and `os` modules. The `sys` module only contains the following symbols:

```
ps1, ps2, copyright, version, platform, exit, maxint
```

The `os` module is reduced to the following functions:

```
error, fstat, listdir, lstat, readlink,
stat, times, uname, getpid, getppid,
getcwd, getuid, getgid, geteuid, getegid
```

Note that restricted code has some read-only access to the filesystem via functions like `os.stat` and `os.readlink`; if you wish to forbid all access to the filename, these functions must be removed.

In restricted mode, there are various attributes of function and class objects that are no longer accessible: the `__dict__` attribute of class, instance and module objects; the `__self__` attribute of method objects; and most of the attributes of function objects, namely `func_code`, `func_defaults`, `func_doc`, `func_globals`, and `func_name`.

The `__import__()` and `reload()` functions are replaced by versions which implement the above restrictions. Finally, Python's usual `open()` function is removed and replaced by a restricted version that only allows opening files for reading.

To change any of these policies, whether to be stricter or looser, see the section below on customizing the restricted environment.

2 Frequently Asked Questions

How do I guard against denial-of-service attacks? Or, how do I keep restricted code from consuming a lot of memory?

Even if restricted code can't open sockets or write files, it can still cause problems by entering an infinite loop or consuming lots of memory; this is as easy as coding `while 1: pass` or `'a' * 12345678901`. Unfortunately, there's no way at present to prevent restricted code from doing this. The Python process may therefore encounter a `MemoryError` exception, loop forever, or be killed by the operating system.

One solution would be to perform `os.fork()` to get a child process running the interpreter. The child could then use the `resource` module to set limits on the amount of memory, stack space, and CPU time it can consume, and run the restricted code. In the meantime, the parent process can set a timeout and wait for the child to return its results; if the child takes too long, the parent can conclude that the restricted code looped forever, and kill the child process.

If restricted code returns a class instance via `r_eval()`, can that class instance do nasty things if unrestricted code calls its methods?

You might be worried about the handling of values returned by `r_eval()`. For example, let's say your program does this:

```
value = r_env.r_eval( expression )
print str(value)
```

If `value` is a class instance, and has a `__str__` method, that method will get called by the `str()` function. Is it possible for the restricted code to return a class instance where the `__str__` function does something nasty? Does this provide a way for restricted code to smuggle out code that gets run without restrictions?

The answer is no. If restricted code returns a class instance, or a function, then, despite being called by unrestricted code, those functions will always be executed in the restricted environment. You can see why if you follow this little exercise. Run the interpreter in interactive mode, and create a sample class with a single method.

```
>>> class C:
...   def f(self): print "Hi!"
...
```

Now, look at the attributes of the unbound method `C.f`:

```
>>> dir(C.f)
['__doc__', '__name__', 'im_class', 'im_func', 'im_self']
```

`im_func` is the attribute we're interested in; it contains the actual function for the method. Look at the function's attributes using the `dir()` built-in function, and then look at the `func_globals` attribute.

```
>>> dir(C.f.im_func)
['__doc__', '__name__', 'func_code', 'func_defaults', 'func_doc',
 'func_globals', 'func_name']
>>> C.f.im_func.func_globals
{'__doc__': None, '__name__': '__main__',
 '__builtins__': <module '__builtin__'>,
 'f': <function f at 1201a68b0>,
 'C': <class __main__.C at 1201b35e0>,
 'a': <__main__.C instance at 1201a6b10>}
```

See how the function contains attributes for its `__builtins__` module? This means that, wherever it goes, the function will always use the same `__builtin__` module, namely the one provided by the restricted environment.

This means that the function's module scope is limited to that of the restricted environment; it has no way to access any variables or methods in the unrestricted environment that is calling into the restricted environment.

```
r_env.r_exec('def f(): g()\n')
f = r_env.r_eval('f')
def g(): print "I'm unrestricted."
```

If you execute the `f()` function in the unrestricted module, it will fail with a `NameError` exception, because `f()` doesn't have access to the unrestricted namespace. To make this work, you'd must insert `g` into the restricted namespace. Be careful when doing this, since `g` will be executed without restrictions; you have to be sure that `g` is a function that can't be used to do any damage. (Or is an instance with no methods that do anything dangerous. Or is a module containing no dangerous functions. You get the idea.)

What happens if restricted code raises an exception?

The `rexec` module doesn't do anything special for exceptions raised by restricted code; they'll be propagated up the call stack until a `try...except` statement is found that catches it. If no exception handler is found, the interpreter will print a traceback and exit, which is its usual behaviour. To prevent untrusted code from terminating the program, you should surround calls to `r_exec()`, `r_execfile()`, etc. with a `try...except` statement.

Python 1.5 introduced exceptions that could be classes; for more information about this new feature, consult http://www.python.org/doc/essays/stdexceptions.html. Class-based exceptions present a problem; the separation between restricted and unrestricted namespaces may cause confusion. Consider this example code, suggested by Jeff Rush.

t1.py:

```
# t1.py

from rexec import RHooks, RExec
from t2 import MyException
r= RExec( )

print 'MyException class:', repr(MyException)
try:
    r.r_execfile('t3.py')
except MyException, args:
    print 'Got MyException in t3.py'
except:
    print 'Missed MyException "%s" in t3.py' % repr(MyException)
```

t2.py

```
#t2.py

class MyException(Exception): pass
def myfunc():
    print 'Raising', 'MyException'
    raise MyException, 5

print 't2 module initialized'
```

t3.py:

```
#t3.py
import sys
from t2 import MyException, myfunc
myfunc()
```

So, 't1.py' imports the `MyException` class from 't2.py', and then executes some restricted code that also imports 't2.py' and raises `MyException`. However, because of the separation between restricted and unrestricted code, `t2.py` is actually imported twice, once in each mode. Therefore two distinct class objects are created for `MyException`, and the `except` statement doesn't catch the exception because it seems to be of the wrong class.

The solution is to modify 't1.py' to pluck the class object out of the restricted environment, instead of importing it. The following code will do the job, if added to `t1.py`:

```
module = r.add_module('__main__')
mod_dict = module.__dict__
MyException = mod_dict['MyException']
```

The first two lines simply get the dictionary for the `__main__` module; this is a usage pattern discussed above. The last line simply gets the value corresponding to 'MyException', which will be the class object for `MyException`.

3 Customizing The Restricted Environment

3.1 Inserting Variables

While restricted code may be completely self-contained, it's common for it to require other data: perhaps a tuple listing various available plug-ins, or a dictionary mapping symbols to values. For simple Python data types, such as numbers and strings, the natural solution is to insert variables into one of the namespaces used by the restricted environment, binding the desired variable name to the value.

Continuing from the examples above, you can get the dictionary corresponding to the restricted module named `module_name` with the following code:

```
module = r_env.add_module(module_name)
mod_dict = module.__dict__
```

Despite its name, the `add_module()` method actually only adds the module if it doesn't already exist; it returns the corresponding module object, whether or not the module had to be created.

Most commonly, you'll insert variable bindings into the `__main__` or `__builtins__` module, so these will be the most frequent values of `module_name`.

Once you have the module's dictionary, you need only insert a key/value pair for the desired variable name and value. For example, to add a `username` variable:

```
mod_dict['username'] = "Kate Bush"
```

Restricted code will then have access to this variable.

3.2 Allowing Access to Unrestricted Objects

Often, the code being executed will need access to various objects that exist outside the restricted environment. For example, an applet should be able to read some attributes of the object representing the browser, or needs access to the `Tkinter` module to provide a GUI display. But the browser object, or the `Tkinter` module aren't safe, so what can be done?

The solution is in the `Bastion` module, which lets you create class instances that represent some other Python object, but deny access to certain sensitive attributes or methods.

`Bastion``(`*object, [filter], [name], [class]* `)`

Return a `Bastion` instance protecting the class instance *object*. Any attempt to access one of the object's attributes will have to be approved by the *filter* function; if the access is denied an `AttributeError` exception will be raised.

If present, *filter* must be a function that accepts a string containing an attribute name, and returns true if access to that attribute will be permitted; if *filter* returns false, the access is denied. The default filter denies access to any function beginning with an underscore '_'. The bastion's string representation will be `<Bastion for` *name*`>` if a value for *name* is provided; otherwise, `repr(`*object*`)` will be used.

class, if present, would be a subclass of `BastionClass`; see the code in 'bastion.py' for the details. Overriding the default `BastionClass` will rarely be required.

So, to safely make an object available to restricted code, create a `Bastion` object protecting it, and insert the `Bastion` instance into the restricted environment's namespace.

For example, the following code will create a bastion for an instance, named `S`, that simulates a dictionary. We want restricted code to be able to set and retrieve values from `S`, but no other attributes or methods should be accessible.

```
import Bastion
maindict = r_env.modules['__main__'].__dict__
maindict['S'] = Bastion.Bastion(SS,
          filter = lambda name: name in ['__getitem__', '__setitem__'] )
```

3.3 Modifying Built-ins

Often you'll wish to customize the restricted environment in various ways, most commonly by adding or subtracting variables or functions from the modules available. At a more advanced level, you might wish to write replacements for existing functions; for example, a Web browser that executes Python applets would have an import function that allows retrieving modules via HTTP and importing them.

An easy way to add or remove functions is to create the `RExec` instance, get the namespace dictionary for the desired module, and add or delete the desired function. For example, the `RExec` class provides a restricted `open()` that allows opening files for reading. If you wish to disallow this, you can simply delete 'open' from the `RExec` instance's `__builtin__` module.

```
module = r_env.add_module('__builtin__')
mod_dict = module.__dict__
del mod_dict['open']
```

(This isn't enough to prevent code from accessing the filesystem; the `RExec` class also allows access via some of the functions in the `posix` module, which is usually aliased to the `os` module. See below for how to change this.)

This is fine if only a single function is being added or removed, but for more complicated changes, subclassing the `RExec` class is a better idea.

Subclassing can potentially be quite simple. The `RExec` class defines some class attributes that are used to initialize the restricted versions of modules such as `os` and `sys`. Changing the environment's policy then requires just changing the class attribute in your subclass. For example, the default environment allows restricted code to use the `posix` module to get its process and group ID. If you decide to disallow this, you can do it with the following custom class:

```
class MyRExec(rexec.RExec):
    ok_posix_names = ('error', 'fstat', 'listdir', 'lstat', 'readlink',
      'stat', 'times', 'uname')
```

More elaborate customizations may require overriding one of the methods called to create the corresponding module. The functions to be overridden are `make_builtin`, `make_main`, `make_osname`, and `make_sys`. The `r_import`, `r_open`, and `r_reload` methods are made available to restricted code, so by overriding these functions, you can change the capabilities available.

For example, defining a new import function requires overriding `r_import`:

```
class MyRExec(rexec.RExec):
    def r_import(self, mname, globals={}, locals={}, fromlist=[]):
        raise ImportError, "No imports allowed--ever"
```

Obviously, a less trivial function could import modules using HTTP, or do something else of interest.

4 References

See some of the papers on the Knowbot Programming Environment on CNRI's publications page: "Knowbot programming: System support for mobile agents", at http://www.cnri.reston.va.us/home/koe/papers/iwooos-full.html, and "Using the Knowbot Operating Environment in a Wide-Area Network", at http://www.cnri.reston.va.us/home/koe/papers/mos.html.

For information on Java's security model, consult the Java Security FAQ at http://java.sun.com/sfaq/index.html.

Perl supports similar features, via a software package called Penguin developed by Felix Gallo. Humberto Ortiz Zuazaga wrote a paper called "The Penguin Model for Secure Distributed Internet Scripting", at http://www.hpcf.upr.edu/ humberto/documents/penguin-safe-scripting.html. Thanks to Fred Drake for bringing it to my attention.

Work has also been done on Safe-Tcl; see "The Safe-Tcl Security Model", by Jacob Y. Levy, Laurent Demailly, John K. Ousterhout, and Brent B. Welch, in the Proceedings of the 1998 USENIX Annual Technical Conference. Usenix members can access the paper online at http://www.usenix.org/publications/library/proceedings/usenix98/levy.html.

The Janus project provides a secure environment for untrusted helper applications by trapping unsafe system calls. The project page is http://www.cs.berkeley.edu/ daw/janus/. Thanks to Paul Prescod for suggesting it.

Can you suggest other links, or some academic references, for this section?

5 Version History

Sep. 12, 1998: Minor revisions and added the reference to the Janus project.

Feb. 26, 1998: First version. Suggestions are welcome.

Mar. 16, 1998: Made some revisions suggested by Jeff Rush. Some minor changes and clarifications, and a sizable section on exceptions added.

Oct. 4, 2000: Checked with Python 2.0. Minor rewrites and fixes made. Version number increased to 2.0.