

# Controlling Access to Resources Within The Python Interpreter

Brett Cannon  
University of British Columbia  
brett@python.org

Eric Wohlstadter  
University of British Columbia  
wohlstad@cs.ubc.ca

## Abstract

Version 2.5 of the Python programming language contains no mechanism for restricting access to resources by Python code. This is a slight hindrance to the language as it is used in many situations, such as a domain-specific language in other applications, where some mechanism to control what resources Python code can access would be helpful.

Python did once have a security mechanism for restricting resource access, but it was disabled in version 2.3. The disabling of the security mechanism was driven by a lack of security expertise on the part of the Python development team. This means that any introduced security mechanism should, if possible, not require language support so as to prevent the need to turn off any new security mechanism in the future.

This paper presents a security mechanism whose impact upon the Python language is minimal. By removing four function or methods from Python's built-in namespace and utilizing Python's modularity in terms of its connection with its underlying interpreter, the proposed security mechanism has minimal impact upon the language. The mechanism allows for controlling access to resources within a single Python interpreter. This allows Python to have some form of a security mechanism between Python code and the system it is running on.

## 1. Introduction

The Python programming language [32] has a wide range of uses. It is used for teaching computers science [27], as a domain-specific language (DSL) [31], and for developing large applications [6]. Part of this popularity stems from the fact that Python is an interpreted language with its own virtual machine.

Something that all users of Python have in common is trust that the code they are executing in the Python interpreter is not malicious. The current version of Python (version 2.5) lacks any mechanism to enforce a security policy related to controlling access to resources (as defined by [11]). While some informal methods exist, Python lacks any mechanism built into the language or its implementation to prevent code from accessing resources such as files or sockets.

The lack of a way to restrict resource access is unfortunate because having a way to enforce a security policy for the Python interpreter would be useful in several situations. When Python is embedded in an application as a domain-specific language (DSL), it

would be helpful to make sure that users of the application can feel confident that some source code they download from the Internet for use will not delete files from their file system that it should not have access to. In the educational realm making sure students do not accidentally harm their own computers would help alleviate worries of a programming mistake causing serious damage.

For these reasons and various other ones, this paper addresses work done to introduce the needed mechanisms to control access to resources from within the Python interpreter. The contribution of this paper is to present an approach to introducing a mechanism into Python that allows for controlling access to resources that does not require changes to the Python language itself and only minor changes to some select object types.

## 2. The Python Programming Language

The Python programming language is an object-oriented, interpreted language that has uses in a wide range of domains such as scientific applications, web sites, and desktop applications [5]. Systems programming is essentially the only area where Python is not currently in use.

Python supports both the procedural and object-oriented programming paradigms along with some functional support. Python's embrace of object-oriented programming is deep. Within Python everything is an object; numbers, strings, functions, etc. There is complete support for operator overloading by objects. There is also deep support for introspection capabilities on objects. For instance, every object stores a reference to the class or type that the object is an instance of (stored in the `__class__` attribute).

One of the basic tenants of Python is that "Explicit Is Better Than Implicit" [26]. This has led to Python being designed so that most actions taken within the language have an explicit representation through some particular object. The semantics are also explicit. A good example of this explicitness is the promotion within Python of modularity amongst source files.

In Python a file that contains source code is called a module. It is a common practice to reasonably separate source code into individual modules, where semantics makes such a separation reasonable. In order to bring a module into another module's namespace an explicit statement must be made using the `import` keyword. Specifying a module with the `import` keyword "imports" the module such that the code in that module is then available to the code that executed the import (e.g., "`import foo`" brings the `foo` module into the current namespace and makes the objects defined within the module accessible under the `foo` name).

But what makes the `import` keyword a wonderful example of Python's explicit modularity is how the statement works. The mechanism that implements the `import` keyword for interpreting a file and creating an object representation is handled by the `--import--` function. Defined in the built-in namespace that is accessible by all executing code, the function is passed the name of

the module being requested for importation and returns the object representation of that module: `__import__('foo')`”.

The `import/__import__` dichotomy also serves as a good illustration of how Python’s interpreted nature ties into the language. Python itself exploits the fact that it is an interpreted language by allowing certain language semantics to be configurable based on the interpreter as it is running. The interpreter is not a blackbox that one cannot change but a part of the larger system that can be modified in certain ways to play a role in how Python code is executed.

By providing a rich C API to support tying into the Python interpreter, Python has garnered great success as a DSL in various programs implemented in C. The C API also allows for modules to not only be implemented in Python source code but also in C.

### 3. History of Security in Python

Currently Python lacks a mechanism for enforcing a security policy; this was not always the case. Added to Python in 1995, the `rexec` module [7] (along with the related `Bastion` module introduced in 1996) provided the ability to run source code in a sandbox. Following the design of `Safe-Tcl` [23], `rexec` used Python’s `exec` statement to run code with a restricted built-in and global namespace. Support was also included in the implementation of Python’s interpreter in the form of flagging execution frames that had a differing built-in namespace from that of the interpreter as being in a “restricted” mode. This restricted mode turned off certain introspection capabilities deemed dangerous. This mechanism was also available at the C level through Python’s C API.

The `rexec` module provided the means of enforcing a security policy when executing Python code for many years. But starting in version 2.3 of Python, both the `rexec` and its companion `Bastion` module were disabled. This stemmed from the key fact that there were (and still are) not any security experts among the various developers of Python.

Because Python is an open source project the only people who help develop Python are volunteers. An issue that stems from Python’s open source development is there is no way to guarantee that any member of the development team is an expert in security as someone would need to volunteer to fulfill that role. Having no security expert on the development team can cause issues as someone might make a change to the language that breaks security but is not caught by anyone on the development team. The realization that the security mechanism in place was being unintentionally broken happened during the development of Python 2.3 and led to the disabling of `rexec` and `Bastion` until a developer joined the development team who had the proper expertise on security [33].

### 4. Related Work

The work presented by this paper is obviously not the first attempt to introduce a mechanism to enforce a security policy within a programming language. As stated in Section 3, `Safe-Tcl` [23] was the inspiration for Python’s first security mechanism. The approach taken in `Safe-Tcl` is to allow multiple interpreters to execute within the same process. Each interpreter has a security policy enforced by controlling what commands are available in the global namespace. By creating the proper delegates an interpreter can be provided commands that have restricted abilities as the programmer deems fit.

Perl provides two types of security. One is a taint mode on all string inputs [25]. Tainting provides for information-flow security by flagging all incoming data as “tainted” [36]. This allows Perl’s interpreter to disallow tainted data to be used in any place that is defined unsafe (e.g., writing to a file or socket, command to the operating system, etc.).

The other type of security in Perl is the `Safe` module [10]. The module restricts the namespace that code will be run in. The code being restricted is then compiled to only execute within the namespace specified to restrict the abilities of the code in question.

`CaPerl` is a modified version of the Perl language that implements a capability system within the language [18]. By introducing new keywords into the language that specify what is trusted and what is not one is able to restrict what abilities are exposed to various pieces of code. It is based on the ideas of capabilities which are discussed later in Section 5.3.

Ruby’s security is much like Perl’s taint system [30]. The significant difference between Ruby and Perl, though, is that Ruby implements levels of security for its taint model. The taint levels allow certain operations to allow tainted data while others don’t based on how restrictive the security level is meant to be.

JavaScript provides several means to securely run code when used in a web browser [3][9]. One is using access control to enforce the security policy. By using access control lists (ACLs) a script is restricted to accessing web pages that are listed in that code’s ACL. JavaScript as run in a web browser can also have signed code. This provides guarantees that if you trust the person who signed some code that it was unmodified and thus as trustworthy as the person who signed it.

Java has had its security implementation evolve over the life of the language. Originally, Java followed a sandbox model where a class loader enforced a local namespace so that untrusted code could not influence other code [14]. The sandbox model did not allow for fine-grained control and thus was changed [17]. With Java 2, a security policy was possible to be set as represented by the `java.security.Policy` object. Security domains were also introduced with each class being a member of a single domain. This allowed each object to be connected to the security policy that was associated with the security domain. To enforce this, the execution stack is traced to make sure that the call will not violate the security policy [8]. Java code can also be signed like JavaScript for use in a web browser so as to allow execution of code from trusted sources without worrying about malicious attacks [13].

The .NET framework builds off of Java’s general design and attempts to fix issues that have crept up over the years with Java’s security system [24]. Like Java, .NET verifies that the file format of the code to be run is safe. While both use policies defined externally to the the respective runtimes, .NET uses the intersection of groups of policies specified. Both use an analysis of the execution stack to make sure that a call is safe. .NET can do some static security policy enforcement, unlike Java. There is also the ability to sign code to verify it comes from a trusted source.

Programming languages are not the only way to provide security. Operating systems must also specify security policies on multi-user systems. Probably one of best known instance of an operating system’s security mechanism is FreeBSD’s “jail” facility [16]. By altering the view of the system a user has, the jail facility helps to prevent a user from accessing certain resources. This alteration of view even affects the root account for the jail [19]. Further enforcement is through proper user checks in system calls that can perform possibly malicious actions on behalf of the user.

## 5. Design

### 5.1 Criteria

From the outset, the design of the security mechanism to be implemented for Python has two design constraints. The first one is that the language could not substantially change in terms of feel. Within the Python community, the term “Pythonic” embodies this idea. Anything deemed Pythonic is considered to follow the general philosophies of the Python programming language. While an

extremely nebulous definition, the closest to an “official” definition of the term can be found in [26]. Adding static type checking, for instance, would not be Pythonic as it goes against Python’s promotion of interface typing.

The restriction of not introducing any features to the language that could be deemed not Pythonic has meant that any change visible at the language must first be deemed Pythonic. This is critical in order for current Python programmers to have psychological acceptability for the security mechanism and are comfortable with it [28][35]. If the security mechanism deviates too much from common Python practices it would be rejected by current users, severely hindering its acceptance.

The second restriction placed upon the design is that it cannot hinder Python’s development team. This stems from lessons learned in Python’s history with security as discussed in Section 3. The developers of Python should not have to shoulder a great burden in order to continue to support any security mechanism that is designed and implemented. By making it a requirement of the design that the Python developers do not have to worry about security in every decision they make, there are two benefits. One is that the developers are allowed more freedom in changing the language; minimizing any inconvenience or hindrance lets Python grow and evolve more easily. Second, by not forcing the developers to constantly make sure that their work won’t break any security mechanism, the possibility of a vulnerability being introduced becomes minimized. This helps prevent a reoccurrence of security holes like what happened with the `rexec` module.

An unfortunate side-effect of these design criteria is that deeply integrated security solutions are not acceptable. This makes the solutions taken by Java and .NET not directly applicable in this situation as both languages were designed with security in mind and thus started with their security mechanism integrated into the implementation. Both Java and .NET take an approach where a class loader is directly integrated into their interpreter and that is not acceptable in this instance as that would burden the Python development team, something that this design is trying to avoid.

## 5.2 Threat Model

The purpose of the work under discussion is to handle a threat model where arbitrary Python code is run in a single Python interpreter without fear that it is able to gain access or control of resources it is not explicitly given (Python supports running multiple interpreters within a single process, but this work only addresses the case of a single interpreter running). In terms of what kind of adversary the security mechanism is meant to guard against, there are two. One is where an adversary misrepresents what a piece of Python code does in terms of accessing system resources such as files or sockets. A trusting user then runs the code from the adversary which then accesses resources that the code was not expected to access. The access can include reading, writing, or destroying resources. The accessing could be purely to destroy other resources or could be to steal sensitive information.

The other kind of adversary is one where a programmer happens to accidentally destroy a resource. This might come in the form of a person new to programming accidentally overwriting a file when they meant to open for reading or to write to a new file path instead of an existing one.

With those types of adversaries being considered, the Python interpreter cannot gain access to resources that the process the interpreter is running in possesses without explicitly being given access to those resources. The operating system bestows and restricts the resources a program can access upon individual processes. This makes the process, from the program’s point of view, the powerbox; the object that has the greatest amount of power and that retains the power to give out abilities and resources [29]. If the powerbox did

not regulate what resources the interpreter had at its disposal then it could do anything the Python process could do.

Related to the process acting as the powerbox, the interpreter cannot gain access to the resources of the operating system without explicitly being given them. This issue is mediated by having to go through the powerbox.

In the end, this work is trying to regulate the authority the Python interpreter has upon various resources that can possibly be granted to it by the operating system. By “authority” we mean the “effects a program may cause on objects it can access, either directly by permission, or indirectly by permitted interactions with other programs” [20]. This means we are trying to mitigate the threat posed by code run in a Python interpreter by controlling what resources are exposed to and by the Python interpreter.

## 5.3 Object-Capabilities

In general there are two major types of security for controlling access to resources, “list-oriented” and “ticket-oriented” approaches to security, which are more commonly known as ACLs and capability systems, respectively [28]. The former uses a list of authorized users to control access to resources. Code that is running would be represented as a user, and if the represented user is not listed as authorized to use a resource it would be denied access. A capability system uses tokens or tickets to represent the right to access a resource. When a ticket is presented to gain access to a resource then the code is granted or denied access based on whether that ticket was valid for approving the right to use a resource.

An analogy one might use is trying to gain access to a night club where a special event is occurring. With ACLs, a bouncer at the door would have a list of people authorized to attend the event. By presenting identification establishing you are who you claim, the bouncer can check your name against the list and either permit or block your entrance. A capability system would be based on invitations that every attendee was sent. If someone came to the door and presented a legitimate invitation they would be granted access.

A type of capability system that exists is object-capabilities [22]. This type of security model as implemented in an object-oriented programming language “uses the reference graph as the access graph” [20]. Authority to use a resource is granted based on whether a reference to an object that possesses the authority to access the resource is available. This makes object-capabilities’ security based on the controlling of references to objects.

In order for object-capabilities to work, three things must be in place in the language: references cannot be forged, immutable shared state, and private namespaces [22]. Unfortunately Python only has the first of the three abilities by default within Python itself. The latter two have solutions and are addressed in Section 5.4.

With the basic conditions met for using object-capabilities, the question becomes why object-capabilities over another approach? The answer comes from Python’s dynamic, modular design along with not wanting to modify the Python language if it can be avoided. Object-capabilities can be implemented on a per-object instance. A single function can implement object-capabilities if so desired without requiring every function to use the system as long as the three required features of object-capabilities is met. This case-by-case implementation approach allows for only parts of a system to bother with security. This works well in the situation under discussion as it minimizes what needs to be changed within Python. No huge re-architecture or addition of security needs to be added to Python but only to individual objects where security matters.

And as object-capabilities allows for a per-object implementation it also alleviates tying into the language directly if it is not desired. In the case of ACLs, the language itself needs to understand

ACLs in terms of identities and the proper propagation of identities. In both Java and .NET, for instance, the security mechanism is directly integrated into the interpreters of the two systems. However object-capabilities requires no such thing if the three implementation requirements are met. A function can implement its own token system and decide itself how it will handle those tokens. It is optional for a language to participate in the security mechanism with object-capabilities.

In general there are two problems commonly believed to exist with any capability system along with one weakness [11]. One perceived problem is that of confining capabilities so that they do not improperly propagate (e.g., authority is given to an object that then hands to another object where it is not desired that the second object gain the authority being passed around). In general there is a known solution [21], but in the case of the threat model being considered in this paper it is not an issue. As there is only a single interpreter there is no one to improperly propagate authority to. The powerbox which gave the interpreter the authority in the first place is the only other party even under discussion and the powerbox already has all authority that the interpreter has and then some. There is no other interpreter that could be improperly given authority for something by some other interpreter.

Another perceived problem of a capability system is that of the revocation of rights. This is a consideration for our threat model. If the powerbox decides to revoke the authority to access a resource after it has been granted (e.g., a one-time access to a resource) there needs to be a way to revoke that right. A solution exists through delegating authority through an intermediary [21]. As will be discussed in Section 5.4, there is a way to implement a delegate in Python, albeit not in Python source code but in C code.

Finally, the weakness that capabilities have is in answering the question, “given an object, what subjects can access it” [11]? But in the case of our threat model, this is once again not an issue. As we have only a single interpreter to consider within the powerbox there can only ever be a single subject whose authority is being restricted. This singularity of only having a single interpreter running makes the plurality of “subjects” meaningless in regards to the question being posted and thus a non-issue.

#### 5.4 Securing a “Bare” Interpreter

As a starting point for discussing what must be done to secure Python, consider what could be deemed a “bare” interpreter. When the interpreter has been initialized and no code has been executed within the interpreter it can be called “bare”. If the interpreter cannot control what resources are exposed within the interpreter in this bare state then there is no chance of regulating access to resources as a bare interpreter is essentially the initial state the interpreter is in when it begins executing code.

With no code imported, an interpreter starts off with the built-in namespace created. The namespace contains various things considered essential or highly useful for almost all Python programs. Exceptions, the various built-in data types, and useful functions are contained within the built-in namespace. Controlling this namespace, along with the global namespace of running code, is important as capability systems in general rely on controlling namespaces [15].

Because the built-in namespace is exposed in all Python code, it must be made safe such that objects in the namespace that indiscriminately give access to resources are removed or modified. Types must either be changed so that one cannot instantiate new instances indiscriminately from them or be removed completely. Methods on the various types in the built-in namespace might need to be removed or modified if they have the authority to access resources that should be protected. The same possibility or modification or removal applies to functions. Table 1 lists the various

methods and functions that need to be removed from the built-in namespace.

In terms of functions, the `execfile` and `open` functions need to be removed. The former takes a file path to a Python source file and executes it in the interpreter, returning an object representing the code that was executed. This is dangerous as it allows access to the file system indiscriminately. The `open` function has the authority to open files for reading or writing. For the same reason as `execfile` needs to be removed, the `open` function needs to be taken out of the built-in namespace; it has authority to access the file system indiscriminately. It is possible to provide a delegate to allow for restricted use of these functions; such a delegate is discussed in Section 6.3.

Two data types in the built-in namespace that need to be changed in order to prevent indiscriminate instantiating of them are the `code` and `file` types. For both types the way to prevent instantiation is to remove their `__init__` methods as this is the method Python calls on a class or type when it creates a new object instance.

The `code` type allows one to create code objects, which represent executable Python code. The type needs to restrict indiscriminate instantiation as it is possible to create code objects with Python’s bytecode without going through Python’s internal compiler. Since Python lacks any way to verify the soundness of bytecode this becomes a possible denial-of-service (DoS) attack by crashing the interpreter. An escalation of abilities might be possible through some very specifically crafted bytecode, although this is pure speculation.

The other type that must be prevented from being directly instantiated is the `file` type. As the object representation of files on the file system, unchecked instantiating of the type would lead to uncontrolled access of the file system. By removing the ability to instantiate the `file` type directly all file access can be redirected through a delegate (such as a delegate for the `open` function). This provides a single access point for the file system within the built-in namespace which makes for simpler control of file system access.

In a bare interpreter, there is a single method that causes Python to not meet the immutable shared state requirement of object-capabilities (as mentioned in Section 5.3). The `__subclasses__` method on the `object` type returns a list of all subclasses. By virtue of the `object` type being the root type of all objects in Python the `__subclasses__` method returns *all* classes defined by any Python code within the Python process, not just within the interpreter as some objects are created to allow the interpreter to work. Without the removal of this method it would not be possible to prevent access to a class or type.

#### 5.5 Securely Allowing Imports

The usefulness of code that runs in a bare interpreter is fairly limited. Without the ability to import external code almost all useful functionality would need to be re-implemented by the code running in the interpreter. Obviously not being to import code would be a great hindrance on programs and cripple the usefulness of Python.

A basic overview of importing modules is covered in Section 2, but a more thorough discussion is called for to understand how imports need to be changed. There are five different kinds of modules that vary based on how the code is implemented and stored. One is built-in modules which are implemented in C and compiled into the Python executable. “Frozen” modules are Python bytecode stored in C code that is then compiled into the executable. Extension modules are written in C and compiled into external object files. Python bytecode (which typically has a `.pyc` or `.pyo` file extension) are byte-compiled versions of Python source code. Python source modules are written entirely in Python source code (and typically has a `.py` file extension).

To remove	Type of object	Purpose of removal
<code>object.__subclasses__</code>	method	Exposes all defined classes. Removal creates immutable shared state.
<code>open</code> <code>file.__init__</code>	function method	Allow unmitigated read and write access to files. Also exposes information about existence of files.
<code>execfile</code>	function	Allows access to any Python source file. Can also be used to query about existence of files.
<code>code.__init__</code>	method	Can create malicious bytecode which is never verified to be well-formed.

**Table 1.** Built-in methods and functions to remove from Python’s built-in namespace.

Regulating what modules can be imported is critical to controlling what resources Python code has access to. With a bare interpreter a safe environment is provided. By controlling imports the environment can stay safe for use as imports are the only way to introduce code that has authority to perform actions that are not available within a bare interpreter.

Regulation of what modules may be imported is done based on the kind of module. Python source code and bytecode modules, in isolation, have only the abilities provided by a bare interpreter. Modules implemented in C, though, can have the same abilities as C code, which is essentially the ability to do anything. By blocking or carefully controlling modules that are implemented in C code then only those modules that can be trusted can be allowed to be imported and thus regulate what abilities the interpreter has access to.

Controlling built-in, frozen, and extension modules is critical to prevent unmitigated access to executing C code. A whitelist should be implemented for each type of module so that only those modules deemed safe through a security audit may be imported. By making the whitelist on a per-type basis instead of based on name alone allows alternative implementations of a module’s interface to be implemented by another kind of module (e.g., implementing an extension module’s interface in pure Python source code).

Python bytecode files should not be allowed for any reason. They are considered an optimization and not a requirement for the Python interpreter to be able to import Python source modules. They pose a possible risk through ill-formed bytecode by either crashing the interpreter or gaining escalated rights. Theoretically trusted bytecode should only be written to the file system by the Python interpreter, but other programs could be running on the same system that attempt to create ill-formed bytecode and trick the interpreter into trusting the bytecode.

With imports properly controlled so that potentially dangerous modules cannot be accessed, and the bare interpreter considered safe, any Python source code should be trusted to run safely. The bare interpreter provides a secured base for any pure Python source code. By restricting imports the protected base can be safely extended through modules without losing its security.

## 5.6 Discussion

In what has become considered a seminal paper on computer security [28], eight design principles are laid out that any security mechanism should strive to follow.

First, one should drive to keep a design simple: “economy of mechanism”. Overall the design presented in this paper is straightforward. Making the bare interpreter safe consists removing some functions or methods. The greatest complexity is adding support for regulating import based on the kind of module being requested. But since the import machinery is an integral part of the Python language it is simple to test any modifications by running all of Python’s regression test suite with any changes implemented. Compared to `rexec`, the new design is simpler since no direct support is needed within the interpreter.

“Fail-safe defaults” dictates that what is to be allowed by a security policy should be explicit, not implicit based on what is **not** allowed. By excluding potentially dangerous types and functions from the built-in namespace, the bare interpreter by default is protected. And by using whitelisting instead of blacklisting for imports the default security is to deny instead of allow importing possibly dangerous modules.

By having “complete mediation” all access to objects need to be checked for authority. Object-capabilities mediate this to reference access. Because there is no explicit check of authority, this principle does not directly apply. Indirectly, object-capabilities performs a “check” every time a reference is used since references cannot be forged in Python and thus any use of a reference is an implicit clearance to use it.

Because Python is open source software, the language itself has an “open design”. The security mechanism proposed in this paper does not rely on any hidden value or implementation detail. Auditing of the security mechanism is possible for anyone and is encouraged.

There is no need for a “separation of privilege” with this security mechanism. There is only a single interpreter in a single process being considered. The concept of multiple owners of a Python process does not exist. This also applies to the principle of “least common mechanism”.

The concept of “least privilege” is left up to the user through the whitelisting of modules. But by using whitelisting over blacklisting users are not given overly broad privileges by accident.

“Psychological acceptability” should be very high for this design. No new syntax or concept has been introduced into Python. The greatest change is that of whitelisting of certain type of modules. While this is a change to how Python normally operates, it will manifest itself in the normal error of an import error. Whitelisting could be viewed as specifying what is installed in the interpreter and verifying a program can run is like checking installation requirements; nothing new to programmers.

## 6. Implementation

Since the security work occurs outside of the interpreter, most of the implementation details are in C code. As such, unless otherwise noted, any code mentioned relating to implementation details should be considered in C code.

### 6.1 Built-In Functions

Removing the built-in functions `execfile` and `open` is straightforward. Python’s built-in namespace is a dictionary stored on the C struct that represents the interpreter (i.e., the `builtin` field of the `PyInterpreterState` struct). Removing the functions from the dictionary is a simple call to the `PyDict_DelItem()` function to delete the items from the dictionary.

### 6.2 Built-In Types

To understand how the `file` and `code` types have been altered one needs to understand how Python implements types at the C level. In C code the `PyTypeObject` struct represents a Python type. Its

various fields represent internal data along with functions that are to be exposed in Python as methods. Two such functions are those that are stored in the `tp_new` and `tp_init` fields and are exposed in Python as `__new__` and `__init__` methods. The `tp_new` function is used to allocate the memory for any new instance of the type along with putting the instance in a valid state. The `tp_init` function is what takes the arguments passed in during the instantiating call to the type and creates the proper state within the instance based on those arguments.

For both the `file` and `code` types the `tp_init` fields in `PyObject` have been set to `NULL`. This is the equivalent of removing the `__init__` methods from the types. Leaving the `tp_new` fields alone for both types allows for either type to be instantiated but not be initialized in any specific way that exposes the authority to access resources.

Both types need a way to initialize new instances within C code, though. For `code` objects the `capiPyCode_New()` function already exists for this purpose. For the `file` type, the `PyFile_Init()` function has been introduced. This allows the `open` built-in to Python, at the C level, to continue to create and initialize `file` instances by calling `PyFile_Init()` after allocating the instance.

### 6.3 Import

For implementing the changes required for the `__import__` function to protect the `import` statement, the code implementing `__import__` was completely rewritten. As it currently stands within Python the `__import__` function is implemented entirely in C. While this is not bad in and of itself, it does raise the level of complexity for working with the code. In order to introduce the required flexibility into the implementation of `__import__`, two design decisions were made: to re-implement `__import__` in Python and to use importers and loaders as defined in Python Enhancement Proposal (PEP) 302 as much as possible [34].

The decision to re-implement `__import__` in pure Python code was a practical decision. There was no specific requirement that we not work with the existing C implementation. It was decided that it would be more expedient to rewrite the function than trying to work with the existing C code in order to make the second design decision we made easier.

Our second design decision to follow PEP 302 is for flexibility reasons. PEP 302 specifies a system that allows for flexible control over how modules are imported. PEP 302 builds off of Python's idea of a search path, much like the `PATH` environment variable on UNIX operating systems. Stored at `sys.path`, it is a list of strings specifying places for the import machinery to look for modules.

PEP 302 extended this simple approach in two keys ways. One is the introduction of an API for importers and loaders. Importers are objects that implement an interface to be queried as to whether a path entry can handle the requested import. Loaders are objects that handle the actual loading of the module. There is machinery in place to associate each entry on `sys.path` with an importer so that each location can be handled individually. Implementing `__import__` in terms of PEP 302 in this regard means separating the aspects of finding and loading modules into objects for the various kinds of modules that Python can import. This provides better modularity and control over how different types of modules are handled. It is also different from the original state of affairs where a monolithic function call at the C level handled the importation of modules.

The other key way PEP 302 enhances how Python imports modules is the introduction of a meta path. While `sys.path` works for modules that have a filesystem location, not all modules have a specific location. Modules that are built into Python do not have a sense of location, for instance. The addition of `sys.meta_path`

provides a path similar to `sys.path` such that modules that have no filesystem location can be included in the search path for modules.

With an implementation of `__import__` that follows PEP 302 the needed level of modularity is available to whitelist modules based on the kind of the module being requested. An importer and loader for built-in modules and frozen modules has been written that resides in `sys.meta_path`. For pure Python source code, bytecode, and C extension modules, an importer and loader that can be configured to handle any of the three mentioned types of modules has also been written that handles path entries on `sys.path`. All importers and loaders can either be left out of the system so as to completely block the importation of a specific kind of module (as with Python bytecode modules) or to place a whitelist delegate in front of an importer and loader (as for built-in, frozen, and C extension modules).

But how does one prevent the manipulation of the whitelist if Python lacks any form of private namespace for objects? To handle this a two-part solution is used. First, the object called to handle imports is stored in the `sys` module in the `import_` attribute. This can be safely done as the `sys` module is not exposed by default as it is a built-in module. As long as a user does not whitelist `sys` then code running in the interpreter cannot reach the object stored at `sys.import_`.

Second, a simple delegate is implemented in C that when called passes its arguments to the object stored at `sys.import_`. By implementing the delegate in C, `sys.import_` is in no way exposed as discussed in Section 5.4.

One issue that came up during implementation is how best to handle modules that are cached during startup of the interpreter. The Python interpreter stores all imported modules in a Python dictionary stored at `sys.modules`. The cache allows future imports to return quickly from the cache. It also provides consistency by having all modules use the same instance of a module instead of a newly created instance for each import. Because the cache is kept in a module that is blocked from being imported by default it is not a security risk itself. The issue, though, is how to not expose modules that must be imported for the interpreter to function, and thus cannot be deleted, but should not be exposed by default (e.g., the `sys` module).

The solution used is to hide all modules that are required for the interpreter to function but are deemed dangerous to expose by default to be put into a Python dictionary that is stored in `sys.modules` under the `.hidden` key. The `__import__` function is changed such that it does not ever allow the importation of a module that has a leading `“.”`. By hiding modules based on name it allows returning modules from the cache to be no more expensive than a simple string prefix check on the module that is being imported.

Placing a capability on all modules that are safely imported and checking for that capability when returning from the cache was considered as a solution to this problem. The current solution was chosen instead however, because it allows for providing an alternative implementation of a module that is hidden. If the capability approach was used then whether a module with a specific name could ever be imported would be an all-or-nothing proposition because name clashes would prevent an alternative module. With the selected solution that is not the case as the name of the module being hidden is not directly used in the cache.

## 7. Current Status

At the time of writing the implementation is not complete [1]. Both the `file` and `code` types have been modified. The `__import__` function has been rewritten in Python source code. The modules required for Python to run but should not be exposed have been moved into the `.hidden` key in `sys.modules`. Proper preven-

tion of directly trying to import `.hidden` has been implemented. `sys.modules` is also cleared of all unessential modules.

What has not been implemented is a proof-of-concept application where Python is used as a DSL. Because this has not been done some changes cannot be made on the testing interpreter as the build toolchain for Python includes a Python script that needs access to some functionality that is considered a security risk but runs with the newly built interpreter. The `open` and `execfile` have not been removed because of this build dependency. Whitelisting of modules has been implemented in the `__import__` function but it has not been integrated into the interpreter. Once the proof-of-concept application is written it is trivial to remove the mentioned built-in functions and integrate the whitelisting.

In terms of vetting the design presented in this paper, it has been presented in three separate venues. The first is the e-lang mailing list [2] where various people involved in the object-capabilities world discuss object-capabilities languages. The Python development team [4] is the second venue this work has been presented. Finally, at the PyCon 2007 Python community conference [12] a talk was given on the work. In all three venues no objection to the design was brought up and it received favourable reviews.

## 8. Future Work

Two directions where this work can go is to introduce a private namespace within the Python language itself and trying to extend the work to act as a replacement for the `rexec` module. In terms of introducing a private namespace for Python, it would remove the necessity to utilize the barrier between Python and C exclusively to implement delegates.

One possible way to provide a private namespace is through closures. Python contains support for closures where free variables can be accessed in a read-only manner (Python 2.6 will allow for read-write access to free variables). But closures cannot be used as a private namespace as they are currently implemented because the values of free variables are exposed through introspection (as of Python 2.5). If the introspection was removed then closures would be usable for delegates when written properly. Figure 1 has an example implementation of how removing introspection abilities on free variables could allow a security delegate for `open` to be safely implemented in Python code.

For re-implementing the `rexec` module, one possible approach is to utilize Python's ability to have multiple interpreters in a single process. By providing a module that allowed for executing code in a separate interpreter it may be possible to replicate the `rexec` module's abilities. It would require analysing what objects are shared between interpreters and making sure there are no covert channels. Analysis would also be needed to see how returned objects from the code executed in a restricted interpreter cannot gain escalated rights in the creating interpreter.

## 9. Conclusion

This paper has presented a design for a mechanism to restrict access to resources by code running within a Python interpreter. Thanks to the modularity of the Python interpreter and the use of object-capabilities, no changes of the language is necessary to allow a certain level of security for Python code when using a single interpreter. Only minor changes or removal of key functions, methods, and types in the built-in namespace is needed that are directly viewable by running code. And with a redesign of the import machinery, proper protections are complete for preventing access to things such as files and sockets.

Several things could have prevented this solution from working. If the built-in namespace was not modifiable then any dangerous built-ins would have ruined any security. Had Python not used an

```
from os.path import join, normpath

def protected_open(open_fxn, restricted_dir,
                  path_join, path_resolver):
    '''Return a closure that restricts the use of
    files to a specific directory.

    Parameters:
    * open_fxn
        Use to open files.
    * restricted_dir
        Directory that files are to be restricted to.
    * path_join
        Function that joins parts of a path together.
    * path_resolver
        *resolves a path (e.g., handles '..').

    '''
    # Define a closure.
    def open(path, flags='r'):
        '''Open a file at 'path' using optional
        'flags'.'''
        # Construct an absolute path.
        joined_path = path_join(restricted_dir, path)
        abs_path = path_resolver(joined_path)
        # Verify the path does not violate the
        # directory restriction that is in place.
        if not abs_path.startswith(restricted_dir):
            msg = ('path leads outside of allowed '
                  'directory')
            raise ValueError(msg)
        # Return the file object.
        return open_fxn(abs_path, flags)

    # Return the closure.
    return open

# In Python 2.5, the value of the 'open_fxn' free
# variable for the open_delegate function can be
# accessed at
# 'open_delegate.func_closure[0].cell_contents' which
# prevents closures to be used as delegates.
open_delegate = protected_open(open, '/tmp/restricted',
                              join, normpath)

# Prevent functions from being exposed through
# 'open_delegate.func_globals'.
del join, normpath
```

**Figure 1.** Example implementation of a security delegate for opening files. Requires that introspection on free variables of closures is not allowed.

exposed function for the `import` keyword then proper protections of external code would not have been possible.

And object-capabilities played a key role as well. Its design allows for using the system in such a way that the Python language did not require modification. By harnessing various parts of how Python's interpreter is implemented the required semantics of object-capabilities could be introduced into Python through the interpreter and not through the language.

## References

- [1] bcannon-objcap branch. Subversion code repository @ svn.python.org.
- [2] e-lang mailing list. <http://www.eros-os.org/mailman/listinfo/e-lang>.
- [3] Javascript security in communicator 4.x. <http://web.archive.org/web/20040621232800/http://developer.netscape.com/docs/manuals/communicator/jssec/index.htm>.
- [4] Python development team. <http://www.python.org/dev/>.
- [5] Python programming language. <http://www.python.org/>.
- [6] Pythonology. <http://www.pythonology.com/>.
- [7] rexec module. <http://www.python.org/doc/2.5/lib/module-rexec.html>.
- [8] Java security overview, April 2005.
- [9] ANUPAM, V., KRISTOL, D. M., AND MAYER, A. J. A user's and programmer's view of the new javascript security model. In *USENIX Symposium on Internet Technologies and Systems* (1999).
- [10] BEATTIE, M. *Safe perldoc page*, perl 5.8.6 ed. Perl Foundation.
- [11] BISHOP, M. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.
- [12] CANNON, B. Securing python. PyCon talk, February 2007.
- [13] DEAN, D., FELTEN, E. W., WALLACH, D. S., AND BALFANZ, D. Java security: Web browsers and beyond. Tech. Rep. TR-566-97, Princeton University, February 1997; 20 Pages.
- [14] GONG, L., MUELLER, M., PRAFULLCHANDRA, H., AND SCHEMERS, R. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems* (Monterey, CA, 1997), pp. 103–112.
- [15] KAMP, P.-H., AND WATSON, R. Building systems to be shared, securely. *Queue* 2, 5 (2004), 42–51.
- [16] KAMP, P. H., AND WATSON, R. N. M. Jails: Confining the omnipotent root. In *In Proceedings of the 2nd International SANE Conference* (2000).
- [17] KOVED, L., NADALIN, A. J., NEAL, D., AND LAWSON, T. The evolution of Java security. *IBM Systems Journal* 37, 3 (1998), 349–364.
- [18] LAURIE, B. CaPerl. <http://caperl.links.org/>.
- [19] MCKUSICK, K. The jail facility in FreeBSD 5.2. *login* 29, 4 (Aug. 2004).
- [20] MILLER, M., AND SHAPIRO, J. Paradigm regained: Abstraction mechanism for access control, 2003.
- [21] MILLER, M., YEE, K., AND SHAPIRO, J. Capability myths demolished, 2003.
- [22] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
- [23] OUSTERHOUT, J. K., LEVY, J. Y., AND WELCH, B. B. The Safe-Tcl security model. *Lecture Notes in Computer Science* 1419 (1998), 217–??
- [24] PAUL, N., AND EVANS, D. .NET security: Lessons learned and missed from java. In *Proceedings of the 20th Annual Computer Security Applications* (2004).
- [25] PERL FOUNDATION. *Perl security perldoc page*, perl 5.8.6 ed.
- [26] PETERS, T. PEP 20: The zen of python.
- [27] RANUM, D., MILLER, B., ZELLE, J., AND GUZDIAL, M. Successful approaches to teaching introductory computer science courses with python. *SIGCSE Bull.* 38, 1 (2006), 396–397.
- [28] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Communications of the ACM* 17, 7 (July 1974).
- [29] STIEGLER, M., AND MILLER, M. A capability based client: The darpabrowser. Tech. Rep. BAA-00-06-SNK, Combex, June 2002.
- [30] THOMAS, D., FOWLER, C., AND HUNT, A. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, October 2004.
- [31] VAN ROSSUM, G. *Extending and Embedding the Python Interpreter*. Python Software Foundation.
- [32] VAN ROSSUM, G. Python language reference. <http://docs.python.org/ref/ref.html>.
- [33] VAN ROSSUM, G. Bastion too (was: Cross compiling). python-dev email, January 2003.
- [34] VAN ROSSUM, J., AND MOORE, P. PEP 302: New import hooks.
- [35] WHITTEN, A., AND TYGAR, J. Usability of security: A case study, 1998.
- [36] ZDANCEWIC, S. Challenges of information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)* (2004).