6.893 Term Project Report: Securely Executing Untrusted Code in Python

Victor Williamson and Stephen Woodrow
{victorw,woodrow}@mit.edu

11 December 2009

1 Introduction

As the popularity of the Python language increases, we expect that it will be used in settings that require execution of untrusted code. For example, Google App Engine uploads and runs third party applications that must somehow or another be sandboxed to prevent access to private data. A similar case could be made for using Python as an application-specific scripting language. We can run all untrusted code into a separate process or interpreter, but that does not scale for performance, and may not be possible in some situations, such as virtual web hosting. An alternative approach is to provide sandboxing at the level of the programming language. Python is not statically typed and all data is globally accessible for public viewing and modification via introspection or other means. This is great for increased flexibility in applications with well defined and programmer-enforced specifications. The goal of this project to examine possible solutions for running untrusted code in Python with minimal impact and via an opt-in policy. Ideally fine-grained access checking on objects is desired, but perhaps not achievable without setbacks in performance and usability. We use Pypy, a Python interpreter written in Python, to prototype our design.

2 Prior art

There is ongoing work to safely run untrusted code in Python. RestrictedPython [5] allows programmers to compile untrusted code with a modified global namespace and with more restrictive implementations of print, getattr, setattr, import, etc. The programmer then runs the untrusted code by passing it to the exec method. It requires the programmer to think carefully about how to fully implement the desired security policy, and precludes modification of or interaction with the untrusted code via attribute access or method calls. The Zope3 [8] package runs untrusted code in an untrusted interpreter where non-basic objects are wrapped in security proxies to mediate access to methods and attributes. Exec statements and try/catch blocks are disallowed. The untrusted interpreter does allow modules running outside of the restricted environment to access variables after code completion but not to call methods or access objects in a piecemeal fashion. In other work [6], a capability-based system can be built in Python using ctype attribute removal or by enabling the Python Interpreter's Restricted Execution that restricts access and modifications made to attributes of Python's builtin objects. This is a work in progress but promises to provide a capability-secure subset of Python similar to what Google Caja [2] found for javascript. Pypy [3] allows trusted code, called an outer controller, to fully virtualize untrusted code by compiling it into a separate executable that channels all IO through stdout and stdin where the controller fully controls IO responses to the untrusted code [4].

3 Threat Model

3.1 Basic Attacker

Imagine a case where a program uploads third party Python modules, dynamically imports the module into the interpreter, and runs the code by invoking its methods. There are expected specifications we expect the third party module to implement, and it has access to Python's global name space. In a root privileged environment malicious code would severely damage the system. Let's instead assume that the uploaded code runs as an unprivileged user with some read access and minute write access on a small number of directories. In this context, we want to defend against modification to other modules that the program depends on. An attacker can import the program modules and library modules to modify any defined attributes and functions that it wishes.

The following are presumed about an attacker.

- An attacker can modify any object in the Python namespace
- An attacker will not loop indefinitely
- An attacker has all the privilege of an unprivileged www user which includes network access and minimal file system permissions
- An attacker encapsulates code within methods and classes that are callable and instantiable from trusted code

3.2 Advanced Attacker

In this case, we wish to go beyond module integrity and consider some of the realistic avenues of attack that an attacker could embed in an arbitrary module under their control that is passed to a Python interpreter and executed via an agreed-upon interface. These avenues include calls to unsafe modules like `os`, introspection, accessing data that is, by *convention*, private/hidden, and importing other modules. Also, we would like to be able to provide untrusted code with some controlled access to potentially unsafe modules like `httplib`, ideally via a fine-grained user-controlled mechanism, to enable use of the rich Python libraries in a safe and predictable manner.

Beyond Step 1, we are interested in broadening the threat model to encompass a larger set of behaviors, ideally all of the following if we can design mechanisms to :

- An attacker may try, by default, to execute arbitrary code on the system, including system (e.g. `os.exec*()`) and network calls, as well as calls from the `inspect` module and access `__builtins__` and other default/global modules and variables that would allow an attacker to inspect and possibly modify objects in the interpreter.
- An attacker may try, by default, to access other objects within the name space, including "private" variables within classes, etc.
- An attacker may try, by default, to import additional arbitrary code or standard modules.

We assume that the attacker is not exploiting vulnerabilities in the Python interpreter. For ease of implementation, will likely assume that the code the attacker runs is limited to a module that is imported into a modified environment and possibly passed a set of needed objects (and perhaps controlled/limited by the user), via a specificed interface, to do its job.

4 Design

4.1 Overview

We approach the problem of running untrusted code in python in three layers. Each layer provides security features to defend against more advanced attacks. The first layer adds support to the interpreter to make copied objects available to untrusted code to prevent malicious modification of attributes in standard library modules such as sys. The second layer adds tokens to enforce access checking at the point of object access to hide objects from untrusted code. The third layer provides modified builtin modules to untrusted code to restrict access to the file system, network, signals and other resources which could compromise the machine.

We demonstrate proof of concept using PyPy, an implementation of Python in Python. On first pass, this gives us a better handle of Python's design and makes it easier to manipulate data structures using Python abstractions without having to deal with reference counting, memory management and Python's performance optimizations. Pypy provides a framework for implementing dynamic languages written as subset of Python called RPython as well as the ability to translate the interpreter directly to binary C and Java bytecode before execution. For the scope of this project, we do not translate Pypy but instead run the bare RPython-based interpreter directly on top of CPython at a performance cost.

4.2 Copied Objects

4.2.1 Justification

We employ copied objects as a first layer of defense against modification of trusted code by malicious code. As an example, the sys module holds information about the execution environment and exposes methods to strongly interact with the interpreter. For example, the interpreter's stdin and stdout can be freely set by executing code, and stdout set to None inhibits all programs from running. Debugging methods setprofile and settrace enable user-specified tracing and profiling on every method call, and can be used to implement a DoS because performance is substantially degraded. The sys module has attributes that store its version and the platform information useful to some programs.

4.2.2 Methodology

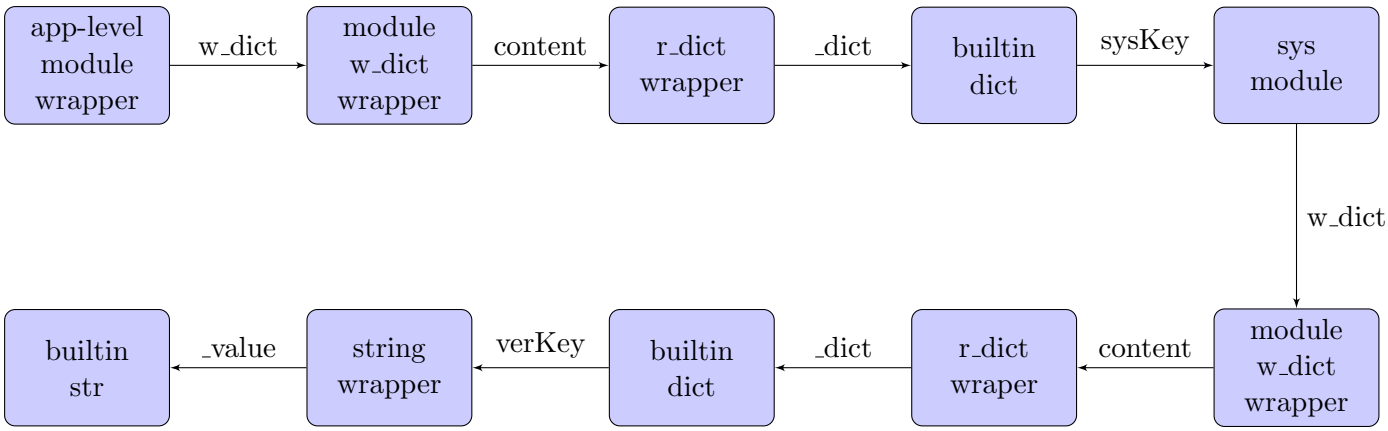
PyPy is divided into interpreter-level and application-level code. Interpreter-level code runs directly on top of Python, while application-level code runs on top of the Pypy interpreter. Some Pypy modules are implemented with both interpreter and application code, and a gateway interface with inter2app and app2interp methods allow calls back and forth. All application-level objects are represented by and wrapped into underlying interpreter-level objects that are distinguished by a 'w_' prefix in the interpreter. To copy a w_ object translates into copying the corresponding application-level object.

To import an untrusted module, the developer suffixes the module name with '_untrusted' in the import statement. The suffix is checked in the interpreter's importhook and a copy of the imported module proceeds. The immediate downside with this approach is that the copy happens after module code has already executed. The threat model for a basic attacker assumes the attacker encapsulates code within methods and classes that are only executed when called and instantiated from trusted code. We ameliorate this threat by using tokens as described in the Tokens section 4.3.

4.2.3 Implementation

At first attempt to copy objects we recursively copy attributes using a depth-first copy. All types, including classes and builtin types such as strings, integers and builtin methods are represented as objects, and this

Figure 1: Module Copy Traversal



approach quickly runs out of stack space because it's not clear when the recursion stops as every object has attributes to recurse on. Our second approach is to implement a breadth-first copy. A breadth-first approach ensures that all mutable objects directly accessible to malicious code are copies of their counterparts used in trusted code, guaranteed to some depth. Our security guarantees are limited without having traversed the attributes of all builtin modules, however, we show that we can achieve important integrity protections by copying less than a dozen levels of attributes.

The time cost to perform breadth-first copy of all attributes on PyPy for interpreter-level w_module objects becomes intractable beyond six levels, taking longer than 15 minutes before we forcefully quit the interpreter. Clearly, the time increases exponentially with the average number of object attributes, which tend to be more than a dozen for builtin objects. However, we can leverage the structure of PyPy w_ representations and restrictions already built into CPython to greatly reduce the amount of copying. First, attributes of CPython's builtin types and functions are read-only and do not have to be copied. These include objects of type object, str, NoneType, builtin_function_or_method, float, int, code, bool, method-wrapper, and instancemethod. Second, only certain portions of w_module objects become available to untrusted code at the application-level. We trace paths to sensitive modules to observe what attributes must be copied, and only copy objects pointed to by the relevant attribute names. Using this approach we can copy a dozen levels of attributes within seconds.

To copy the immediate child attributes of imported modules such as sys it is sufficient to copy w_module objects to eight levels. The interpreter navigates through its wrapped object representations to obtain the actual object that is returned to application level code. Figure 1 shows the attribute traversal path from the wrapped user level module to the sys module version attribute. At the leaf node we copy a W_StringObject that wraps the Python builtin string 'version'. All other sys module attribute pointers have also been copied because the dictionary that holds the attribute mappings has been copied. From this exercise we learn that the relevant attribute names to copy are w_dict, content, _dict and_value.

We actually have to do a little more work because functions execute using the dictionary under their func_globals attribute for namespace lookups. The corresponding attribute we must include in our copies at the interpreter-level is w_func_globals. This does not require copying more than eight levels because we keep track of modules already copied. The sys module is stored in w_func_globals, but has already been copied before being reached so we reuse the existing copy.

The attribute names ending in 'Key' are actually not attribute names but rather keys into Python's builtin dictionary types. For interpreter-level code it has been sufficient to copy the objects pointed to

by keys. However, application-level objects may include keys that are mutable and required being copied themselves. Application-level code may also include container types like list, set, buffer and collections which are sequences that can be enumerated over using the enumerate method, and are handled as a special case in our copy routine.

Special care is taken handling the __dict__ attribute which holds all attribute definitions for modules, classes and instances. Every attribute set on the object during a copy is keyed into the __dict__ dictionary. In Python class attributes can be accessed and modified through object instances. Currently, we depend on the copy module to make shallow copies of our objects, but it does not copy class types, builtin versions of tuple, method, stack trace, stack frame, file, socket, window and other similar types. However, the builtin dir method returns both instance and class attributes which causes our code to update the instance __dict__ with copied versions of class attributes. Directly on CPython this means all class attributes can be modified by untrusted code if it's smart enough to remove its own __dict__ keys for the class attributes. We avoid this problem and many others because PyPy implements classes and other builtin CPython types as plain Python objects in the PyPy interpreter. Tuple elements, for example, are stored in a list container in the PyPy interpreter.

4.2.4 Limitations

The biggest limitation is that our security guarantees for module integrity depend on the depth of the breadth-first copy. We can defend against the common cases, for example, by copying the sys module to a depth of eight. However, we can import the urllib module which also imports sys and a malicious user can access urllib.sys, which can be ameliorated by copying nine levels. If we copy 14 levels we can rule out these most common accesses. Without having exhaustively checked all modules in Python's standard library, we can't rule out that sensitive modules are not traversal beyond 14 levels.

4.2.5 Code

The copy code is found in dist/pypy/bin/victor.py in the fdcop method. We use a helper function fdcoph that is called when each level is reached to copy all attributes at that level. The dcop method is very similar to the fdcop, but does not take into account special _r_dict wrapper keys used in interpreter-level dictionaries. The cop method is a depth-first copy that is not used because of recursion limits. The trace function was used to trace PyPy code and track down import operations. We include a malicious vy.py module that can be imported and has methods to set sys.version, sys.stdin and sys.stdout. When imported as untrusted it only modifies its own copies of sys module attributes. We include the vtrust module to represent some trusted developer who wants to maintain the integrity of his data structures. Start the PyPy interpreter and run the the commands in figure 2 to demo security features.

Figure 2: Module Copy Demo

```
dr-wily:/Documents/6.893/finalproject/securepy/dist/pypy$ python bin/py.py
faking <type 'module'>
PyPy 1.1.0 in StdObjSpace on top of Python 2.5.2 (startuptime: 7.92
secs)
>>>> import sys, vy, vy_untrusted, vtrust
>>>> sys.version
'2.5.2 (69303, Dec 12 2009, 05:31:26)backslashn[PyPy 1.1.0]'
>>>> vtrust.c.l newline
[1, 2, 3]
>>>> vtrust.c.t
(1, 2, 3)
>>>> vy_untrusted.destroy()
trying to destroy your env.
>>>> sys.version
'2.5.2 (69303, Dec 12 2009, 05:31:26)backslashn[PyPy 1.1.0]'
>>>> sys.stdin
<open file '<fdopen>', mode 'r' at 0x09e3476c>
>>>> vy_untrusted.sys.version
'gotcha!'
>>>> vy_untrusted.sys.stdin
>>>> vy_untrusted.hurtvtrust()
Erase vtrust.c.t, append to vtrust.c.l, insert 'trick' attribute to i1.
>>>> vtrust.c.t
(1, 2, 3)
>>>> vtrust.c.l newline
[1, 2, 3]
>>>> dir(vtrust.i1)
['__doc__', '__module__', 'l', 't']
>>>> vy.destroy()
trying to destroy your env.
>>>> sys.version
'gotcha!'
>>>> sys.stdin
>>>> vy.hurtvtrust()
Erase vtrust.c.t, append to vtrust.c.l, insert 'trick' attribute to i1.
>>>> vtrust.c.t
()
>>>> vtrust.c.l newline
[1, 2, 3, 4]
>>>> dir(vtrust.i1) newline
['__doc__', '__module__', 'l', 't', 'trick']
```

4.3 Token-based Object Access Control (TOAC)

4.3.1 Overall Design

The general idea behind this method of access control is to partition Python's globally unrestricted object space into subspaces whose membership is defined by the objects associated with a capability token for that "namespace"[1]. Access by a particular executing module or function to a namespace is controlled by the module's/function's possession of that namespace's token in a special "slot" variable. By controlling which objects have a certain token, and which functions have loaded that same token in an appropriate slot, we can restrict access to objects, which can offer properties of confidentiality and integrity for data objects, and can prevent the access of dangerous functions by untrusted code.

TOAC's underpinnings are implemented at the level of the Python interpreter and stack-based virtual machine. More specifically, TOAC's security guarantee comes from the assumption that if an object cannot be loaded onto the virtual machine's stack, it cannot be accessed or executed against. This principle is leveraged by interposing access control checks on certain operations that load objects onto the virtual machine's stack to prevent an object associated with one namespace from being loaded by another namespace. This approach protects against less obvious means of accessing objects, such as introspection, because it tracks objects, not names or other means of referring to objects.

While this approach to access control may seem overly simple and lacking expressivity for complex policies, the goal of TOAC is not to provide fine-grained access control (i.e. read/write ACLs) on individual objects. Rather, TOAC's goal is allow trusted code to execute untrusted code/modules with:

- Minimal-to-no modification of the untrusted code,
- Confidence that the untrusted code will not jeopardize the confidentiality or integrity of objects in the trusted namespace, and
- Confidence that the untrusted code will not execute dangerous Python functions, such as filesystem and network accessors.

TOAC isn't a replacement for privately-scoped variables and other security mechanisms/approaches in Python, but rather a different approach at providing similar functionality for the particular situation of running untrusted modules or code components (though it could possibly be adapted for some of these purposes).

Our design of TOAC has some nice properties which should make it easy for programmers to take advantage of it's features. All of the capabilities and mechanisms for indicating meaning of capabilities to the interpreter are exposed to the programmer at the application level using existing python language constructs — objects, attributes, and namespace dictionaries. Thus, TOAC should be easy for programmers to take up and use in interesting ways. Also, a number of decisions have been made to minimize or eliminate required changes to existing code (though existing Python code executed as untrusted code may not execute perfectly).

We attempted to minimize the changes to the standard python environment to implement TOAC, and also attempted to have this system "fail gracefully", falling back on safe defaults or intuitive behavior. For example, objects' name tokens can be switched explicitly, but default to inheriting their namespace from their parents. This enables fully transparent operation on existing code as all activity occurs in the same namespace. Further, if the namespace is not partitioned at all, the python interpreter behaves like the stock

[1] "Namespace" is the original term we developed for our approach, which is used here for consistency, but would be changed in future to something more reflective of the actual implementation, such as "object space", etc.

CPython interpreter. These features should theoretically mean that within a particular namespace, most-to-all of the standard Python features (introspection, __builtins__, etc.) should work without modification, though we have not spent much time testing this hypothesis.

4.3.2 Fundamentals & Application-Level Programming Model

In TOAC, capabilities are represented and bound to objects using tokens, and are presented with semantics to the interpreter using "slots". This section discusses these primitive components of TOAC in some detail, and explains how they are exposed at the application layer.

Tokens Tokens are the capabilities of this system, and define the bounds of object access. Each object has a token associated with it, which is used by the PVM access control mechanisms described later. A token is represented by the built-in "nametoken" object type. These are simply opaque objects that have no particular semantic meaning — essentially a clone of "object" with a different type to provide structure because they are presented to the programmer. The unforgeable reference to a particular nametoken instance is what we're interested in. A particular token object doesn't have any special significance; they only become powerful when the token is associated with objects.

Creating Tokens Fresh tokens, for the creation of a new namespace, are created using the __builtins__.newtoken(t function, which instantiates a new nametoken object and inserts it into the current namespace's __alltokens__ dictionary using tokenkey as the dictionary's key, while also returning the new nametoken object. This tokenkey does not affect the creation of the nametoken object itself, but rather is used to allow the user to keep track of the different nametokens it may have in __alltokens__ (which is described later). This step of adding the newly created token to __alltokens__ is necessary because each token is "self-owned" — the token associated with the nametoken object is that same nametoken object.

Assigning Tokens to Objects The token associated with an object can be changed using the __builtins__.changetoken(object,token) function. Restrictions of this function are:

- Token must be a nametoken object — this is a somewhat arbitrary decision, as we only care about the use of an object reference, but it is used to try and provide some consistency for programmer/programming model.
- The frame executing changetoken must hold both object's existing token and the new token — this is to prevent the obvious concern of malicious code hijacking an object, and also prevents honest programmers from shooting themselves in their collective foot.
- A token's ownership cannot be changed — as noted above, tokens are self-owned, and this cannot change. This prevents unintentional loss of control of a token and helps maintain logical consistency — that is, if you hold a reference token, you can access it and all the objects associated with it.

It is worth noting that changetoken() only changes the object passed to it — not any of the object's attributes or other associated objects. This means that changetoken() should be sufficient for changing tokens of primitive objects, but a recursive form of changing tokens for an object and it's attributes/children is necessary for more complex objects. This is not implemented currently, but could possibly be adapted from our object copying code, and would be desirable to implement in future to make it much easier for a programmer to move an object between namespaces.

Assigning Tokens to Modules During Import To reap the greatest benefit from this mechanism, untrusted modules need to be assigned to a namespace as they are imported, both to prevent the code in a module that is executed immediately upon import from causing security problems, but also to intercept and assign tokens properly as the objects within the module are instantiated. The mechanism proposed to accomplish this is the "full" version of the import method, where additional information can be passed as argument into the function call, including an unused dictionary that by default contains the contents of locals(). This saves us the trouble of modifying the Python parser/lexer/etc.

An example might be something like the following, to import a module named "sketchymod":

```
safetoken = newtoken("untrusted")
sketchymod = __import__("sketchymod", globals(), {"__nametoken__": safetoken})
```

This would serve to import the "sketchymod" module into the namespace defined by the "safetoken" token.

While the above import syntax/mechanism is implemented in our changes to Pypy, it causes some problems due to Python's approach of sharing module instances internally between multiple importers. Under the above method of import with a __nametoken__, the first namespace to import a particular module "claims" it by annotating it's instantiated objects with that namespace's nametoken. When other namespaces attempt to import the same module or another module dependent on this initial module, an access check error occurs because the module cannot be shared with another namespace. This problem is an attractive point for future integration with the object copying mechanism discussed previously to work around this problem.

Capability Slots Taking a page from KeyKOS' terminology, a number of token/capability "slots" exist in TOAC. This is the other major application-level component of TOAC. Tokens are loaded in slots to give them special meaning in the eyes of the interpreter, allow the interpreter to check and annotate objects appropriately. Unlike KeyKOS, these slots have specific semantic meaning. These "slots" are not to be confused with Python's __slots__[2].

The current slots are described below. Note that "globals" refers to the dictionary for the global namespace of a particular execution context (module, function, etc.) that is currently loaded into a Python execution frame.

The current slots include:

- `globals.__nametoken__`: This slot contains the primary token for the namespace of currently executing code. When an object is accessed by the executing code, this slot is checked first. When an object is stored by the executing code, it is associated with this token. This slot can also be set using the __builtins__.set_nametoken(token) function.

- `globals.__alltokens__ = {}`: This slot contains a dictionary which in turn contains all of the tokens that the currently executing code can access. The presence of a token in __alltokens__ indicates that the currently executing code can access objects in that token's namespace. However, objects will never be automatically associated with this token — if this is desired, it must be performed explicitly using changetoken().

It would be desirable to implement two additional slots in future versions of TOAC:

- `__builtins__.__universaltoken__`: This slot would be located in the __builtins__ dictionary. It would be used by the interpreter to determine which objects should have universal read/execute access, which will be necessary for certain builtins to function properly while guaranteeing the integrity

[2] `http://docs.python.org/reference/datamodel.html#slots`

of these builtins and avoiding the currently-implemented practice of leaving such objects without any assigned token.

- `proxyfunc.func_closure.__callertokens__ = {}`: The potential addition of this slot is discussed more in the Proxy object section. This slot would be located in the closure of a function that is designated to act as a proxy/wrapper function (generically referred to above as "proxyfunc") for an unsafe function, and would contain a dictionary of tokens representing namespaces that are allowed to call the particular wrapper function. If a calling function's __nametoken__ is not in __callertokens__, then it will not be allowed to execute the proxy function.

4.3.3 Access Control Mechanism & Interpreter-level Implementation

Up to this point, we've mainly discussed the conceptual idea and application-layer programming model for TOAC. Now we describe TOAC's implementation of the access control mechanisms at the Python interpreter level.

As described in the introduction, TOAC's security guarantee is based on the assumption that if an object cannot be loaded on the stack, it cannot be executed against or accessed, preventing "bad" things from being done with that object. This principle is leveraged in the implementation of the access control mechanism by interposing token checks on operations that attempt to load objects onto the Python virtual machine (PVM) execution stack. This guarantee should perhaps be qualified by mention that interpreter-level functions do not execute on the PVM and so have the potential to affect objects without needing to load a reference to said object on the execution stack. This point would require further auditing in the future, but isn't taken up further here, as we are most interested in application-level/user-supplied code.

PVM Opcodes Selected for Interposition To begin the implementation of the access control checks, we first needed to determine which opcodes required access checks, based on our principle mentioned previously. Our primary opcodes of concern are those that load objects onto the stack, but we have other interests and concerns as well. Ultimately, we selected the following criteria — for each opcode, does it's operation:

1. Load an object onto the stack,
2. Delete an object reference, or
3. Store an object from the stack.

An opcode meeting criteria 1 or 2 requires performance of an access check to ensure that the code has the proper token to perform such an action. An opcode meeting criteria 3 requires that the token of the object being stored be updated to the current __nametoken__ (sort of like taint tracking) to confer namespace protection on newly stored or updated objects transparently.

After reviewing the description and implementation of the 143 PVM opcodes based on these criteria, we selected the following 19 opcodes to interpose access checks or token updates upon:

Opcode	Interposing operation
BINARY_SUBSCR	Access check
STORE_SUBSCR	Token update
DELETE_SUBSCR	Access check
STORE_NAME	Token update
DELETE_NAME	Access check
UNPACK_SEQUENCE	Access check
FOR_ITER	Access check
STORE_ATTR	Token update
DELETE_ATTR	Access check
STORE_GLOBAL	Token update
DELETE_GLOBAL	Access check
LOAD_NAME	Access check
LOAD_ATTR	Access check
LOAD_GLOBAL	Access check
LOAD_CLOSURE	Access check
LOAD_DEREF	Access check
STORE_DEREF	Token update
MAKE_FUNCTION	Token update
MAKE_CLOSURE	Token update

A few additional opcodes loaded objects onto the stack by creating objects/importing module objects/etc. However, these were not included at this point because their potential abuse in terms of allowing access to trusted objects appeared to be limited through the restriction of the above opcodes.

4.3.4 Representation of object-token association at the interpreter level

For this proof-of-concept implementation, the association between objects and tokens was implemented at the PyPy interpreter level as a dictionary called nametoken_table. Values in this dictionary were the nametoken for each object, while the key corresponding to particular value was the id()[3] of the object associated with the token. while the value associated with each key was the nametoken associated with each object. This is not ideal for a number of reasons, but it was easy to implement for this first version of TOAC, and the use of id() rather than object references themselves was to allow garbage collection to take place by not holding on to references unnecessarily. Ideally, a future implementation of TOAC would bind the token for an object in the object's data structure itself.

4.3.5 Interposition functions

Two general interposition functions (one for checking if access should be allowed and the other for updating tokens upon storing an object from the stack) were inserted into the implementation of each of the above opcodes as necessary to achieve the desired access check or token update functionality.

Access Check Interposition Function Implemented as namecheck_load() in PyPy's pyopcode.py module, this function accepts the current execution frame and an object to check for access by this current execution frame, and returns a boolean representing whether or not access should be allowed. Thus function begins by checking the given object's token against the __nametoken__ global of the current frame. If they match, access is permitted. If they don't match, then namecheck_load() proceeds to check against

[3]A unique, unforgeable identifying number. In the case of CPython, the memory address of each object.

the __alltokens__ dictionary global of the current frame. If the object's token is not found in __alltokens__ either, then access is denied.

There are two caveats that should be mentioned about the current implementation of namecheck_load(). First, it currently defaults to allowing access in the event that either __nametoken__ or the token associated with an object is not defined. This is an indeterminate state that wouldn't ideally be allowed to occur, and thus would be denied if it were to somehow be encountered. However, as noted later in "Difficulties with PyPy", it was difficult to assign tokens to every object stored by STORE_ opcodes, thus leading to the case where objects are encountered with no associated tokens. Also, this function only checks loads made by Python code compiled into bytecode — it doesn't check functions implemented by the interpreter, which need to be interposed upon separately.

Token Update Interposition Function Implemented as namecheck_store() in PyPy's pyopcode.py module, this function has a much simpler implementation than namecheck_load(). It simply accepts the current execution frame and an object to check for access by this current execution frame, and associates the __nametoken__ of the current frame (if present) with the object.

Responding to access violations When violations to our access control policy due to token mismatches occur, there are two ways of responding, given that we cannot provide the information that was attempted to be accessed. The interpreter can either respond by raising an exception, or by passively returning an object that implicitly signals a failure.

Raising Exceptions The unambiguous and explicit approach to dealing with access violations would be to raise an exception, say an AccessError exception, to notify the code that it has violated the established namespace-based security policy. While this would be effective and reasonably easy to implement, it could pose a problem to existing code that is not equipped to catch such exceptions and deal with them gracefully. Thus, for this current version, we have opted to take a simpler approach in providing a passive response that returns no information but doesn't throw an exception.

Passive response Instead of raising exceptions, we can provide a passive response that is tailored to the type of operation that was attempted, such that the "error" fits more easily into the standard Python development model. For instance, a name-lookup-based opcode will return a NameError exception, suggesting that the name of the desired object was not defined, while a sequence subscript opcode will return a None object instead of the desired object. While this does have the nice property of not requiring existing code to understand and prepare for a new type of exception, this approach may also cause problems in that it has the potential to cause Python to behave in a way that diverges from the Python documentation, such as calls to built-in functions returning a NameError.

It seems that the decision about whether to throw explicit exceptions or provide implicit notice of access control violations is probably best decided by the programmer implementing a TOAC-based system. Thus, we would hope to provide this choice as a programmer-configurable option in a future version of TOAC. There is currently an option available in the namespace_helpers.py module, but it is not yet possible to throw exceptions given the current codebase, and so the use of this option is unadvisable. PyPy also currently outputs access violation messages to stderr to make clear to the user what is taking place when a NameError, etc. is occurring, and this feature is also configurable via namespace_helpers.py

Difficulties with PyPy The use of PyPy as our platform for implementing this proof of concept caused a couple of problems. First, the highly interwoven implementation of the Pypy interpreter coupled with Python's loose typing often made it difficult to determine the proper place to implement a change, and

then to debug unintended faults caused by our changes. Ultimately we succeeded in producing what seems to be a reasonable implementation of several of the features discussed here, but it was a time-consuming process to do so.

Also, and perhaps more seriously, PyPy's use of it's bytecode interpreter to handle bytecode translation to lower-level languages, as well as interpretation of Python bytecode itself, prevented the creation of a "default token" for objects that are attempted to be stored without a corresponding __nametoken__ in the execution frame's globals. The reason why this dual use of the bytecode interpreter presented problems was because of the use of different object types in the translation process that did not allow their globals to be modified. This in turn frustrated initial attempts to implement a more rigorous access check policy as noted above. This problem with applying a default token to all objects created before a token is explicitly instantiated could perhaps be avoided by instead collecting a list of all objects stored by the PVM before the __main__ module is reached, at which point the __main__ module would receive it's default token, and this token would be retroactively applied to all standard objects that were instantiated before this point. Unfortunately, this isn't yet implemented in TOAC.

Brief Thoughts on Performance and Optimization While we didn't have the opportunity to put this access control system through it's paces, it occurs to us that the access control mechanism implementation will be fairly expensive because of it's implementation at the opcode level and the potentially-multiple dictionary lookups required in the event that a token mismatch (access violation) occurs or if the necessary token is located in the __alltokens__ slot. It's possible that opportunistic optimizations could be implemented depending on the intended use of TOAC. We present one example below, which stemmed from discussion with the professor.

If the bulk of the code for a given application were trusted, with only a small component of the code running as untrusted code (i.e. only two namespaces/token spaces), then it would be ideal to prevent checks from occurring on interactions between trusted objects. This could possibly be accomplished by providing two different interpreter-level base objects from which all of the other Python objects were extended. By polymorphism or a simple base class check, objects with the same trusted base class would be exempt from access checks. While this approach seems potentially feasible, it's complete implementation could require significant effort to deal with the details that are not apparent at this point.

4.3.6 Description of Code

The following files were modified to provide the nametoken objects (in `std/`) and the builtin functions to manage them (in `__builtin__/`):

```
/pypy-dist/pypy/module/__builtin__/__init__.py
/pypy-dist/pypy/module/__builtin__/namespace.py
/pypy-dist/pypy/module/__builtin__/namespace_helpers.py
/pypy-dist/pypy/objspace/std/model.py
/pypy-dist/pypy/objspace/std/nametokenobject.py
/pypy-dist/pypy/objspace/std/nametokentype.py
```

`/pypy-dist/pypy/module/__builtin__/importing.py` was modified to incorporate the optional nametoken argument for assignment to the module being imported.

The following files contain the bulk of the modifications for the actual access control checks, including the interposition functions and their actual embedding in the implementation of the many PVM opcodes.

```
/pypy-dist/pypy/interpreter/pyopcode.py
/pypy-dist/pypy/interpreter/function.py
/pypy-dist/pypy/interpreter/nestedscope.py
```

Finally, the demo for object access control and proxy objects consists of the following files (start the demo using nametest.py):

```
/pypy-dist/pypy/nametest.py
/pypy-dist/pypy/nametest_notrust.py
/pypy-dist/pypy/nametest_trust.py
```

4.4 Proxy Objects

4.4.1 Motivation

As we have now established, passing a deep copy of some critical interpreter objects to untrusted code can be sufficient to prevent an attack that disrupts common interpreter resources via the importation and execution of untrusted code. However, there may be objects that cannot be copied that untrusted code may legitimately require access (directly or indirectly) to to perform it's work, such as the builtin open() function that accesses the filesystem. This need calls for an additional type of protection to allow untrusted code to call "dangerous" functions with limitations on the possibility for damage that the untrusted code can perform. We suggest proxy objects will meet this need.

4.4.2 Approach

Our notion of a proxy object within Python is a function that wraps a dangerous function and provides it with some safe limitations and/or defaults specified by the trusted programmer in order to limit the damage of a potentially malicious programmer of untrusted code. This is similar to some of the functions provided during a discussion of namespace management in Java in [7]. As an example in that paper, a file reader that enforces a particular base URL is constructed, similar to the effect of a chroot jail, by interposing a function that "fixes" the pathname as the object is instantiated. Java's strong, static typing system enables this to work easily and without possibility of counteraction by an attacker. Python requires more steps to provide a similar level of protection using closures, an approach proposed by Cannon and Wohlstadter in [1] in an attempt to provide private namespaces in the stock CPython interpreter. An example of such a closure, for providing a chroot-like open() function is provided below:

```
# in trusted code
def f(f_open, base):
    def restricted_open(path, mode):
        f_open(base + path, mode)
    return restricted_open

ropen = f(__builtins__.open,"/jaildir")

# later, ropen is passed to untrusted code, using it as if it were open()
f = ropen("myfile","w")
```

In the example above, the open function from builtins, `f_open`, and the base filesystem path, `base`, are the variables bound in the restricted open closure. Unfortunately, as Cannon and Wohlstadter note, even closures are not safe from Python's introspection capabilities. By accessing `ropen.func_closure[0].cell_contents`, the untrusted code can access the "raw" open() function and proceed to wreak havoc on the filesystem if they choose to do so. If closures could be made private, this proxy system could be feasible. Our previously developed access control approach seems possibly well-suited to this task.

4.4.3 Implementation

The basic proxy object approach described above is implemented in our current software version, and an example of a restricted "open" function is included with our submission. An untrusted module can call a closure-based proxy function that is passed to it, without gaining access to the function's code or closures. The result/return value, by virtue of being placed on the stack by the proxy function without checks, is accessible by the untrusted module. This approach is implemented by simply associating a function with the nametoken of its creator; only the creator namespace can access the function's closure and generally call the function without passing it to another module as a function argument.

A more interesting and feature-full, but also complex, approach that we would be interested in considering for implementation in the future would involve the more rigorous checking of proxy objects and their results being loaded onto the stack, along with the use of the __callertokens__ capability slot discussed earlier. Under this approach, simply having a reference to a proxy function on the stack would be insufficient — the nametoken of a prospective caller would need to be stored in the __callertokens__ capability slot. This approach would allow the proxy function to be made widely available (such as in the builtins, referenceable by all namespaces) while allowing the trusted programmer to narrowly tailor which namespaces can actually call the proxy function. An additional useful feature could stem from more rigorously checking the return value from a proxy function call. By virtue of the proxy function's own nametoken, any new objects returned by it should be associated with that same nametoken. If this value were returned as-is to the untrusted caller and checked strictly upon return, it would not be accessible by the caller because of the nametoken mismatch. This would have the nice property of making proxy function results confidential by default. It would then be up to the proxy function developer to "declassify" the object or appropriate subcomponents for the untrusted caller module's use.

5 Future Work

More research is needed to understand PyPy's standard object space and how their object implementations affect the recursive copying of object attributes. The rate of growth is exponential at each level of breadth-first copy, which has significant time costs when copying large interpreter-level objects. What we'd like to do is use our token mechanism to implement copy-on-write when an object is written to. There are challenges to make this perform well in the face of many memory reads and writes from both trusted and untrusted code.

Additional work would also be valuable to enhance the object access control and proxy object approaches as described above. The development of this "first cut" approach to this system proved to be enlightening in understanding the finer points of the Python execution model, and some changes or improvements, such as the consideration of more opcodes for interposing access checks, would improve the certainty and security guarantee of our system. Further, the basic slot-token system has a number of interesting possibilities for further extension that would both add potentially-interest features and make our security model more complete. It would also be beneficial to audit or consider the interpreter's object accessing functions that have not been thoroughly considered under our stack-focused security model.

More generally speaking, there are potentially great gains to be had from integrating/better integrating the three components of our approach that we studied and developed relatively independently, as noted at points throughout the paper. Integration would ideally provide us with a system that offers the best of a number of approaches: access control would protect the confidentiality and integrity of certain objects, while object copying coupled with access control would allow for creation of a "shared" state per namespace that would allow each namespace to adjust their namespace accordingly without fear of accidental or intentional/malicious damage being done to the operation of the rest of the interpreter. Finally, the use of object proxies, while likely to be initially labor intensive for an implementing programmer, provide a

relatively simple, straight-forward and secure approach to providing necessary access to shared functions or shared state.

6 Conclusion

We developed three interesting approaches for providing separately useful security properties for the handling of untrusted Python code by a trusted interpreter. Object copying allows untrusted code to modify duplicated shared state harmlessly. Token-based object access control allows namespaces to be created that can restrict the object access of certain code/modules to things that cannot be shared or copied, including access through Python features like introspection. Object proxies use object access control to provide untrusted code with safe access to functions and state that cannot be duplicated using the object copying approach. Each of these mechanisms were developed by modifying the Pypy interpreter to some extent, with a focus on attempting to work at the application-level as much as possible.

Each of the three approaches developed in our prototype offers nice features, and we expect that integration would be extremely valuable, given the natural fit of the three approaches as described above. Given further time, we would hope to integrate these mechanisms to allow them to work together seamlessly in terms of behavior, programmer interface, and internal implementation. Altogether, these mechanisms would be helpful in allowing users and programmers to use or include untrusted code in their projects with greater confidence, while maximizing the compatibility of existing code with these new mechanisms.

References

[1] Brett Cannon and Eric Wohlstadter. Controlling access to resources within the python interpreter. http://people.cs.ubc.ca/~drifty/papers/python_security.pdf.

[2] Google caja. http://code.google.com/p/google-caja/.

[3] Pypy documentation. http://code.google.com/p/google-caja/.

[4] Pypy's sandboxing features. http://codespeak.net/pypy/dist/pypy/doc/sandbox.html.

[5] Restrictedpython 3.5.1. http://pypi.python.org/pypi/RestrictedPython/.

[6] Tav. Paving the way to securing the python interpreter. AskTav!, February 2009.

[7] Dan Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. In *In Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, 1997.

[8] Zope 3 api documentation. http://docs.zope.org/zope3/.