Securing Python: *Controlling the abilities of the interpreter*

Brett Cannon
w/ Eric Wohlstadter
PyCon 2007

What is this talk about?

How do you control access to resources by the interpreter for security purposes?

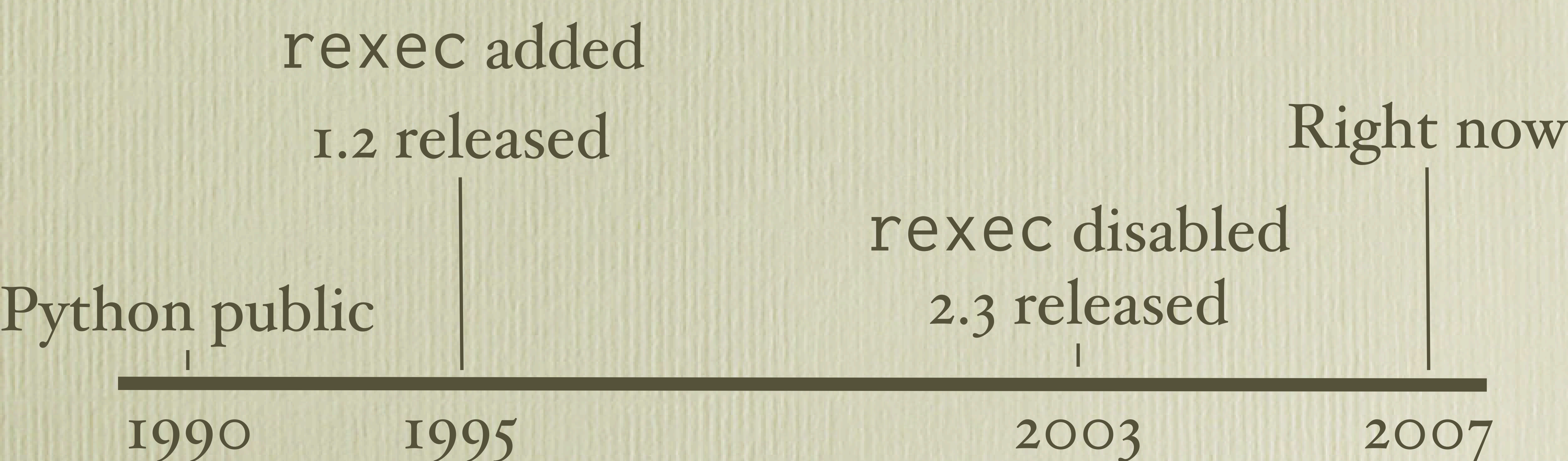
- This talk is NOT about a replacement for `rexec`!
- This talk about controlling the **entire** interpreter as a whole.
- People who embed Python should like this work.
- (Hopefully) can generalize this work into an `rexec` replacement.

A bit of history ...

rexec

Based on Safe-Tcl.
Safe execution of Python code by isolating malicious code and controlling global namespace.

`rexec` Timeline



Why Was `rexec` Disabled?

- Security mechanism dependent on security code in Python's core.
- Not every member of python-dev is a security expert.

Motivation

Originally wanted to use Python for client-side web scripting ...

This meant I wanted to sandbox the entire interpreter to control access to resources.

Design Criteria

- Minimal amount of changes to Python's core.
- No changes to the language if possible.
- Be able to control access to resources by Python code at an interpreter level.

Things I looked At

- **Java** - ClassLoaders and permissions
- **.NET** - Policy domains
- **Ruby** - Taint system
- **Perl** - Safe and taint system
- **Tcl** - Isolation of interpreters
- **JavaScript** - Information flow
- **FreeBSD jails** - User isolation

Discovered there are roughly two types of security ...

Who-I-Am security

Protections based on who you are.

What-I-Have security

Protections based on what you possess.

With the design criteria and possible ways to handle security, I chose ...

Object-capabilities

Don’t play fast and loose with your references!

What *is* that?

A form of
What-I-Have security.

Capability system where an object's reference provides the capability to use something.

Read "Capability Myths Demolished" for nice intro and pretty pictures.

Ka-Ping Yee
is a co-author of the paper.

Needed for Object-Capabilities

1. Immutable shared state
2. Private namespaces
3. Cannot forge references

1.

Immutable shared state.

Don’t have that.

2.

Private namespaces.

Don’t have that either.

3.
References cannot be forged.

At least we have one thing on the list.

So why would I want to use object-capabilities?

It helps minimize the number of changes I need to make to the interpreter.

Python gets to stay "Pythonic" while gaining a way to securely run code.

Compare to ACLs

- Would need a way to differentiate between user code and interpreter-specific code.
- Such a check would probably need integration into the interpreter code itself.
- This goes against design criteria of minimizing changes to core code.

My solution

First, make a “bare” interpreter safe.

“Bare” means no imports have been done beyond what is required to get to an interpreter prompt.

Need to ...

- Make shared state immutable.
- Remove dangerous functions.
- “Neuter” dangerous built-ins.

Immutable shared state.

`object.__subclasses__`

exposes **all** classes/types.

“Neuter” dangerous built-in types.

Remove constructor for `file` and `code`.

Remove dangerous built-in functions.

`open` and `execfile` can be protected by a delegate if needed (explained later).

How do you keep an interpreter safe once you want to import modules?

Need to ...

- Delegate for `__import__`.
- Control imports.
  - Module whitelisting.
  - Deny `.pyc` files.

1.

Utilize the barrier between Python and C.

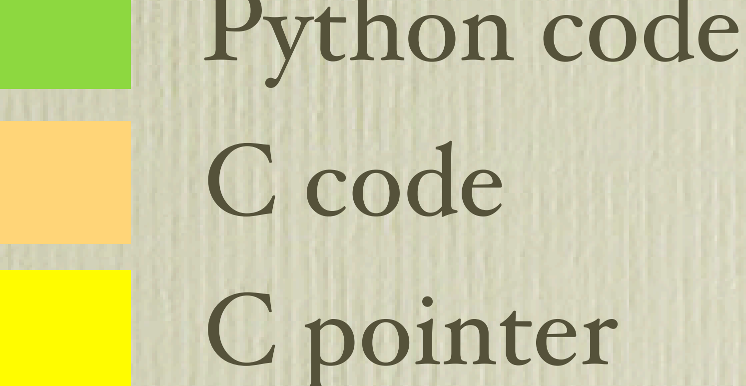
Helps with “private namespace” issue.

Restricted

PyObject

tp_call

PyObject *



2.
Control imports.

If you can't import evil things and you start off lacking access to evil things, you can't be evil, right?

How do you control imports?

Rewrite import machinery in pure Python.

That took a little while.
Neal Norwitz better be happy.

Require whitelisting of modules that could be evil.

If it is written in C, it can be evil.
Python's abilities to do stuff comes from C code.

Perform a security audit before whitelisting something!

No .pyc modules.

Don’t want to have to worry about malicious bytecode that could crash the interpreter or gain escalated privileges.

All Python source files considered safe!

Evil powers from stuff written in C.
If you keep out evil C code, then Python code itself is safe.

Python code can now run safely w/o worry!

Location of code:
`bcannon-objcap` branch

How far along am I?

- Neutered built-ins.
- Import rewritten.
  - Working in interpreter.
  - Whitelisting working in outside of interpreter.

What's Left To Do?

- Proof-of-concept embedding of interpreter.
  - Not easy to build Python when you have turned off various abilities.
- Remove dangerous built-ins.
- Verify whitelisting works in interpreter.

Where can we go from here?

- *Not sure if I will personally get to implement these ideas.*
- Delegates written in Python.
- `rexec` replacement.

Add a private namespace to Python.

Make `__attribute` **truly** private.

Allows Python code to do more security work.

Write a module to run Python code safely in other Python code.

In other words, an `rexec` replacement.
Would use Python's "multiple interpreters, single process" mechanism.

Please email me (brett@python.org) if you have a use for this work!

Looking for use cases for a future research paper.

Thanks

- Eric Wohlstadter
- Neal Norwitz
- Guido van Rossum
- Jeremy Hylton, Alex Martelli, Will Robinson
- Tyler Close, Alan Karp, Mark Miller, Marc Stiegler, Ka-Ping Yee
- Python-Dev and capabilities crowd.

Questions?