CPython byte-code and code-injection

Tom Zickel

Overview

Bytecode and code objects - what are they ?

bytehook - Insert function calls inside pre existing code without preparations.

pyrasite - A way to inject python code into running processes.

bytehook + pyrasite - An experimental way to debug already running servers without previous preparations.

(*) This talk is mostly based on CPython 2 conventions, yet most of the stuff is just a name change in CPython 3.

The problem

```
(pycon)root@theman:~/pycon# cat test.py
import traceback
import random
import time
import os

def computation():
  time.sleep(2)   # YOUR STRANGE AND COMPLEX COMPUTATION
  return random.random()

def logic():
  try:
    res = computation()
    if res < 0.5:
      raise Exception('Low grade')
  except:
    traceback.print_exc()

if __name__ == "__main__":
  print os.getpid()
  while True:
    logic()
```

```
(pycon)root@theman:~/pycon# python test.py
11969
Traceback (most recent call last):
  File "test.py", line 14, in logic
    raise Exception('Low grade')
Exception: Low grade
Traceback (most recent call last):
  File "test.py", line 14, in logic
    raise Exception('Low grade')
Exception: Low grade
Traceback (most recent call last):
  File "test.py", line 14, in logic
    raise Exception('Low grade')
Exception: Low grade
Traceback (most recent call last):
  File "test.py", line 14, in logic
    raise Exception('Low grade')
Exception: Low grade
```

Why bytecode ?

CPython when running python code actually knows how to execute only bytecode.

If we want to modify the code it’s running we need to understand how the bytecode works.

“Bytecode, is a form of instruction set design for efficient execution by a software interpreter. ...bytecodes are compact numeric codes, constants, and references (normally numeric addresses) which encode the result of parsing and semantic analysis of things like type, scope, and nesting depths of program objects…” - Wikipedia

CPython compiles your source code ?

When you type stuff in the interactive shell, import source code, or run the compile command, CPython actually compiles your code.

The output is an **code object**.

It can be serialized to disk by using the marshal protocol for reusability as a .pyc file (projects like uncompyle2 can actually get a .py back from only the .pyc).

The CPython bytecode is not part of the language specification and can change between versions.

The compilation stage is explained in the developer’s guide chapter “Design of CPython’s Compiler”.

What is a code object ?

Code objects represent byte-compiled executable Python code, or bytecode. They cannot be run by themselves.

To run a code object it needs a context to resolve the global variables.

A function object contains a code object and an explicit reference to the function’s globals (the module in which it was defined).

The default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

```
>>> def f(a=1):
...   return a
>>> type(f)           #<type 'function'>
>>> type(f.func_code)    #<type 'code'>
>>> dir(f)
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__doc__',
'__format__', '__get__', '__getattribute__', '__globals__', '__hash__', '__init__', '__module__', '__name__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals',
'func_name']
>>> f.func_defaults
(1,)
>>> f.func_globals
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'f': <function f at
0x00000000022E7D68>, '__doc__': None, '__package__': None}
>>> dir(f.func_code)
['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'co_argcount', 'co_cellvars',
'co_code', 'co_consts', 'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnotab', 'co_name',
'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']
```

Bytecode layout

```
>>> def fib(n):
...     if n <= 1:
...         return 1
...     else:
...         return fib(n - 2) + fib(n - 1)
>>> dis.show_code(fib)
Name:              fib
Filename:          <stdin>
Argument count:    1
Kw-only arguments: 0
Number of locals:  1
Stack size:        4
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: None
   1: 1
   2: 2
Names:
   0: fib
Variable names:
   0: n
```

Line No. Bytecode Index Argument Meaning
Jump Target Opcode Optional argument

```
>>> dis.dis(fib)
  2           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (1)
              6 COMPARE_OP               1 (<=)
              9 POP_JUMP_IF_FALSE       16

  3          12 LOAD_CONST               1 (1)
             15 RETURN_VALUE

  5     >>   16 LOAD_GLOBAL              0 (fib)
             19 LOAD_FAST                0 (n)
             22 LOAD_CONST               2 (2)
             25 BINARY_SUBTRACT
             26 CALL_FUNCTION            1
             29 LOAD_GLOBAL              0 (fib)
             32 LOAD_FAST                0 (n)
             35 LOAD_CONST               1 (1)
             38 BINARY_SUBTRACT
             39 CALL_FUNCTION            1
             42 BINARY_ADD
             43 RETURN_VALUE
             44 LOAD_CONST               0 (None)
             47 RETURN_VALUE
```

How does CPython run the bytecode ?

The interpreter part of CPython takes the function object, and builds a frame object which includes all the information it needs to execute it (the locals/globals/builtins variables, exception information, where you are in the bytecode, etc…)

A huge C function called PyEval_EvalFrameEx takes the code object’s bytecode string, and in a giant switch, for each opcode, it executes its effects.

CPython is a stack based VM.

The GIL can only be switched between opcodes.

bytehook

A python module which allows you to insert calls to functions from other functions via hook points by manipulating the code object.

You can enable / disable each hook point or remove them completely.

My motivation was to insert debug code to live running process without putting pre-existing code to debug them.

I’m using pyrasite to inject bytehook to running processes.

http://github.com/tzickel/bytehook

Disclaimer, this is more of a research project, and is not production ready !

A simple example

```
from __future__ import print_function
import bytehook

def test():
    for i in range(4):
        print(i)

if __name__ == "__main__":
    print('1st try')
    test()

    def odd_or_even(locals_, globals_):
        if locals_['i'] % 2:
            print('odd')
        else:
            print('even')

    hookid = bytehook.hook(test, lineno=2, insert_func=odd_or_even, with_state=True)
    print('2nd try')
    test()
    bytehook.disable_hookpoint(hookid)
    print('3rd try')
    test()
```

```
1st try
0
1
2
3
2nd try
even
0
odd
1
even
2
odd
3
3rd try
0
1
2
3
```

bytehook limitations

It can only modify pure python functions i.e. functions which are not written in C or using C-API.

Modification affects only the next entry into the function, so if you have a loop function which never exits, if you inject a hook into it while it's running, it will not load the changes.

Currently supports CPython 2.7 for now.

Not been thoroughly tested in all situations.

pyrasite

Tools for injecting arbitrary code into running Python processes.

Running python code in a process with python loaded is easy:

- Attached to the target process.
- Acquire the GIL: PyGILState_Ensure()
- Pass your python code as a C char* string: PyRun_SimpleString(code)
- Release the GIL: PyGILState_Release()

(*) easy in GDB, much less easier in Windows (I've ported it to Windows / Mac)

pyrasite payload is simply to exec an .py file.

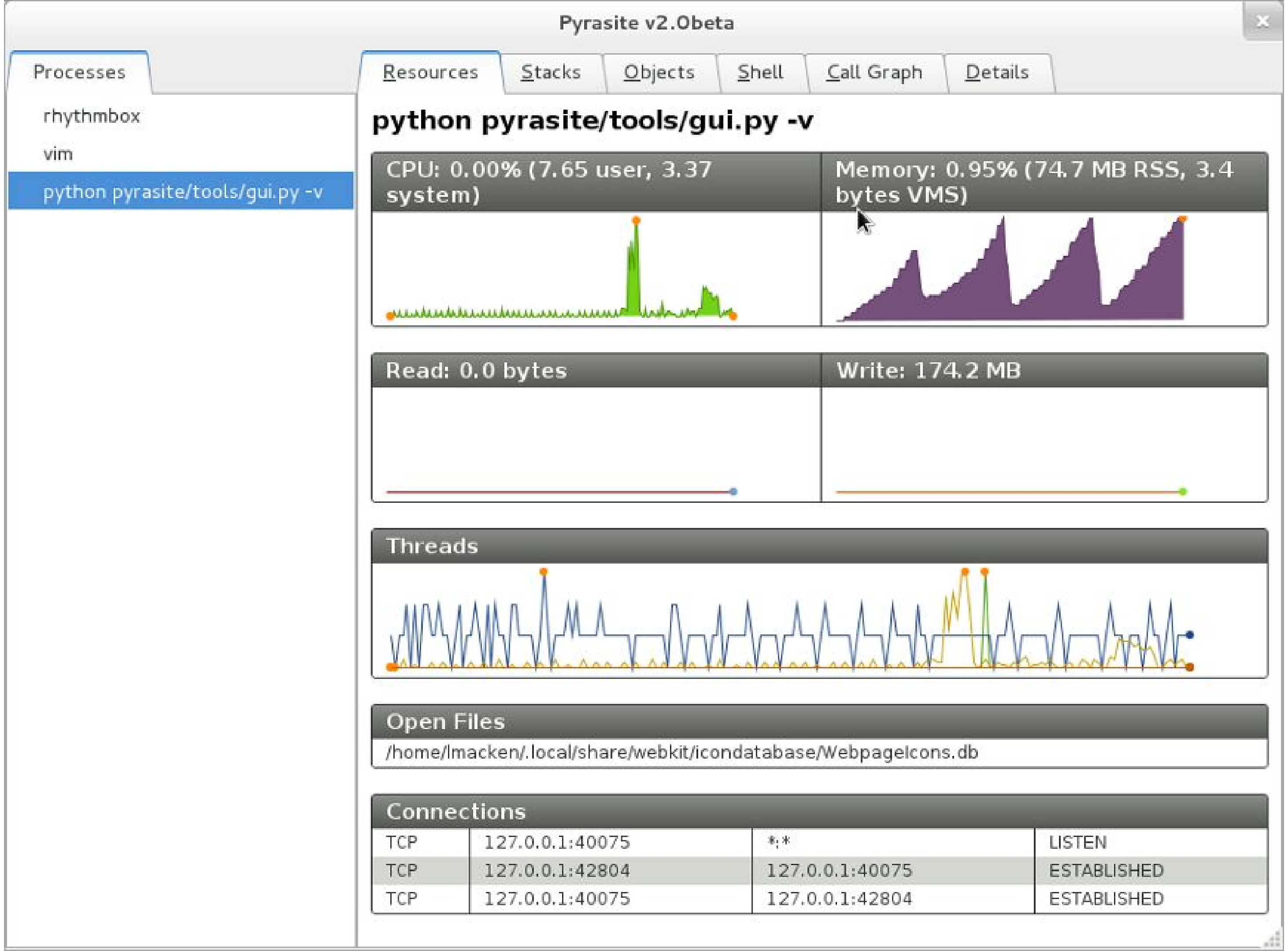
https://github.com/lmacken/pyrasite (started by Luke Macken)

pyrasite

Interesting payloads pyrasite comes with:

- reverse_shell
- dump_memory
- dump_stacks
- dump_modules
- force_garbage_collection
- start_callgraph

Also has a nice GUI (might work in linux only)



rhythmbox
vim
python pyrasite/tools/gui.py -v

python pyrasite/tools/gui.py -v

CPU: 0.00% (7.65 user, 3.37 system)

Memory: 0.95% (74.7 MB RSS, 3.4 bytes VMS)

Read: 0.0 bytes

Write: 174.2 MB

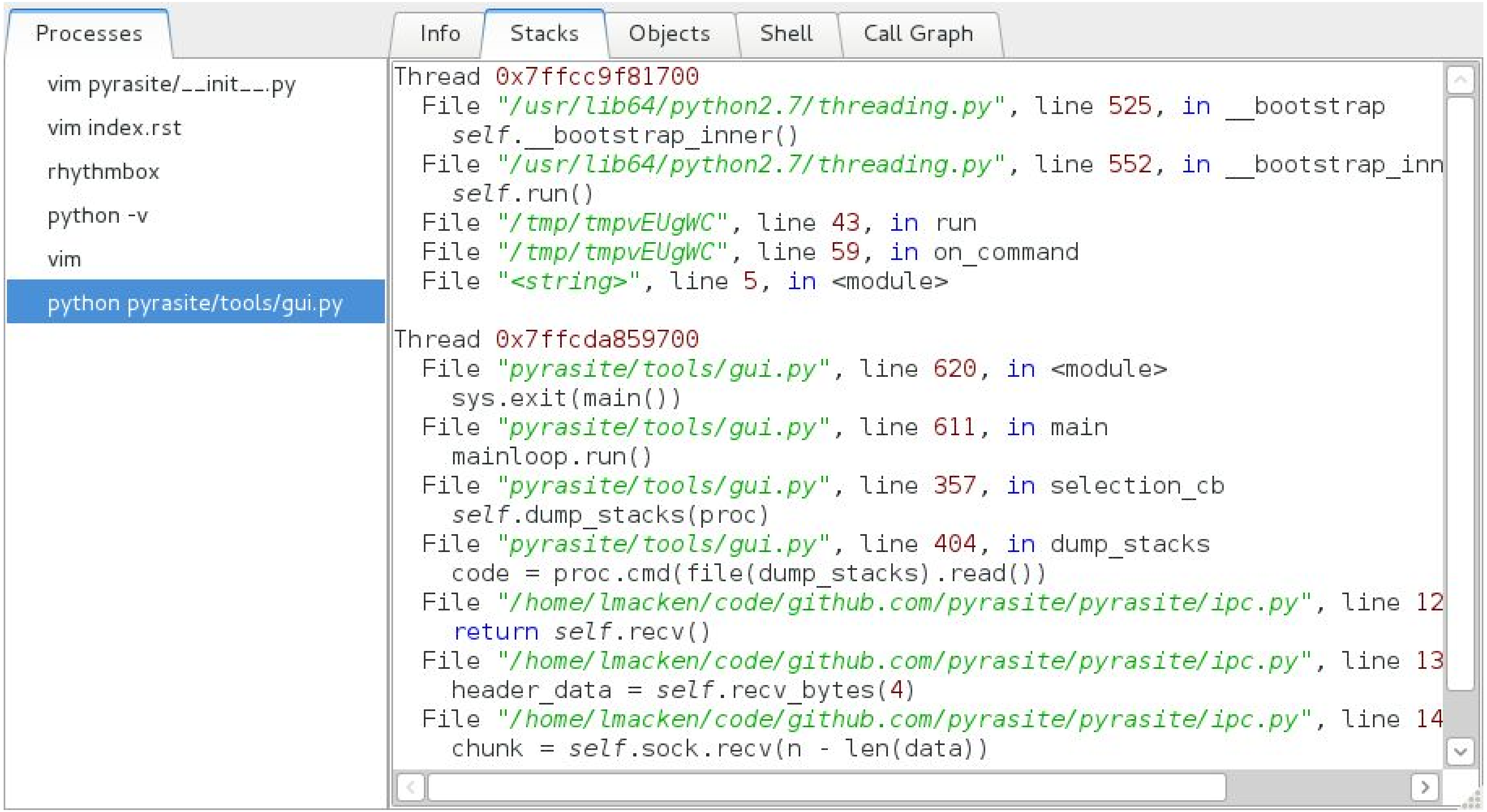
Threads

Open Files

/home/lmacken/.local/share/webkit/icondatabase/WebpageIcons.db

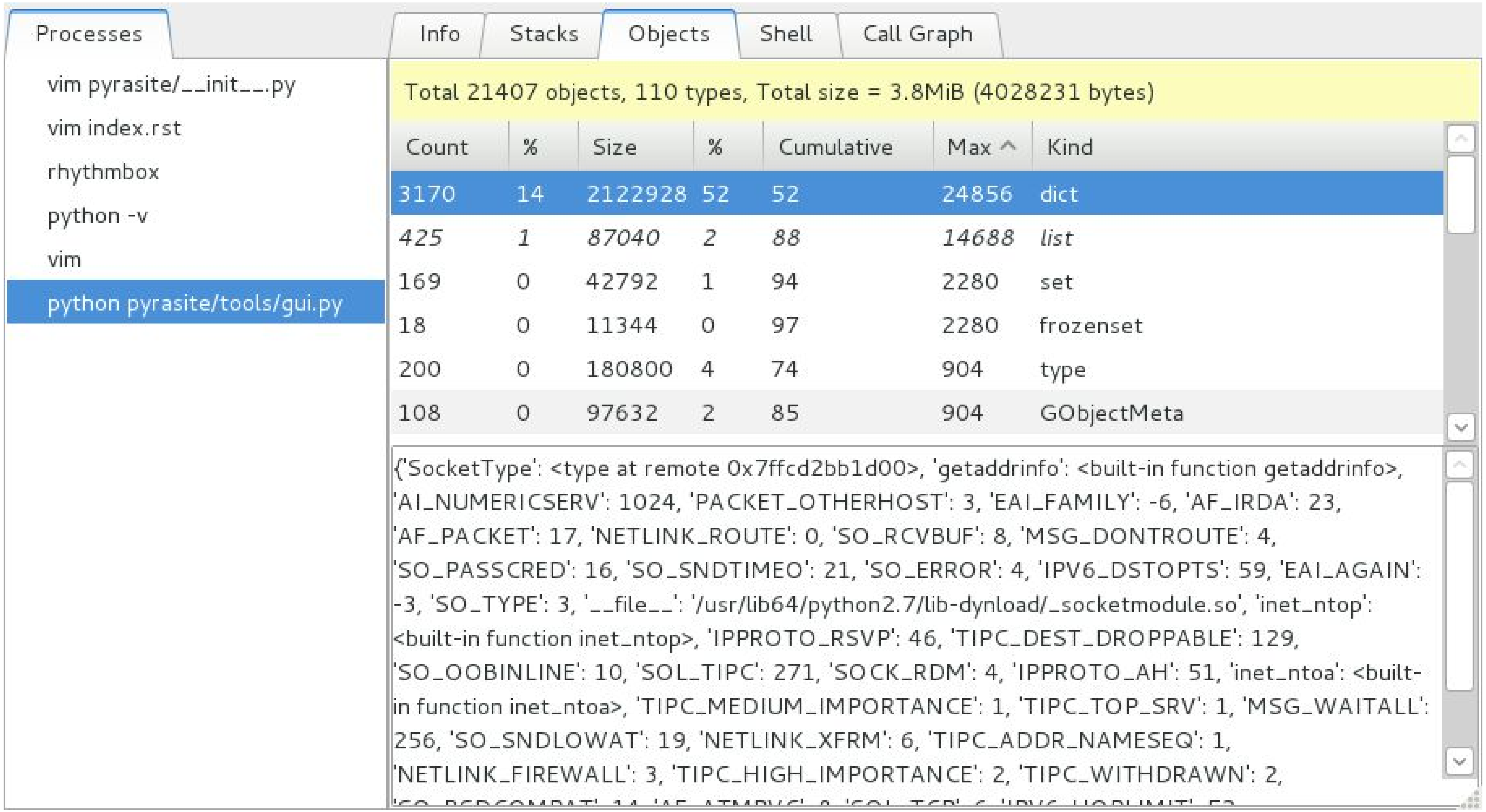
Connections

TCP	127.0.0.1:40075	*:*	LISTEN
TCP	127.0.0.1:42804	127.0.0.1:40075	ESTABLISHED
TCP	127.0.0.1:40075	127.0.0.1:42804	ESTABLISHED



```
Thread 0x7ffcc9f81700
  File "/usr/lib64/python2.7/threading.py", line 525, in __bootstrap
    self.__bootstrap_inner()
  File "/usr/lib64/python2.7/threading.py", line 552, in __bootstrap_inn
    self.run()
  File "/tmp/tmpvEUgWC", line 43, in run
  File "/tmp/tmpvEUgWC", line 59, in on_command
  File "<string>", line 5, in <module>

Thread 0x7ffcda859700
  File "pyrasite/tools/gui.py", line 620, in <module>
    sys.exit(main())
  File "pyrasite/tools/gui.py", line 611, in main
    mainloop.run()
  File "pyrasite/tools/gui.py", line 357, in selection_cb
    self.dump_stacks(proc)
  File "pyrasite/tools/gui.py", line 404, in dump_stacks
    code = proc.cmd(file(dump_stacks).read())
  File "/home/lmacken/code/github.com/pyrasite/pyrasite/ipc.py", line 12
    return self.recv()
  File "/home/lmacken/code/github.com/pyrasite/pyrasite/ipc.py", line 13
    header_data = self.recv_bytes(4)
  File "/home/lmacken/code/github.com/pyrasite/pyrasite/ipc.py", line 14
    chunk = self.sock.recv(n - len(data))
```



vim pyrasite/__init__.py
vim index.rst
rhythmbox
python -v
vim
python pyrasite/tools/gui.py

Total 21407 objects, 110 types, Total size = 3.8MiB (4028231 bytes)

Count	%	Size	%	Cumulative	Max ^	Kind
3170	14	2122928	52	52	24856	dict
425	*1*	*87040*	*2*	*88*	*14688*	*list*
169	0	42792	1	94	2280	set
18	0	11344	0	97	2280	frozenset
200	0	180800	4	74	904	type
108	0	97632	2	85	904	GObjectMeta

```
{'SocketType': <type at remote 0x7ffcd2bb1d00>, 'getaddrinfo': <built-in function getaddrinfo>,
'AI_NUMERICSERV': 1024, 'PACKET_OTHERHOST': 3, 'EAI_FAMILY': -6, 'AF_IRDA': 23,
'AF_PACKET': 17, 'NETLINK_ROUTE': 0, 'SO_RCVBUF': 8, 'MSG_DONTROUTE': 4,
'SO_PASSCRED': 16, 'SO_SNDTIMEO': 21, 'SO_ERROR': 4, 'IPV6_DSTOPTS': 59, 'EAI_AGAIN':
-3, 'SO_TYPE': 3, '__file__': '/usr/lib64/python2.7/lib-dynload/_socketmodule.so', 'inet_ntop':
<built-in function inet_ntop>, 'IPPROTO_RSVP': 46, 'TIPC_DEST_DROPPABLE': 129,
'SO_OOBINLINE': 10, 'SOL_TIPC': 271, 'SOCK_RDM': 4, 'IPPROTO_AH': 51, 'inet_ntoa': <built-
in function inet_ntoa>, 'TIPC_MEDIUM_IMPORTANCE': 1, 'TIPC_TOP_SRV': 1, 'MSG_WAITALL':
256, 'SO_SNDLOWAT': 19, 'NETLINK_XFRM': 6, 'TIPC_ADDR_NAMESEQ': 1,
'NETLINK_FIREWALL': 3, 'TIPC_HIGH_IMPORTANCE': 2, 'TIPC_WITHDRAWN': 2,
```

pyrasite + bytehook

```
(pycon)root@theman:~/pycon# cat test.py
import traceback
import random
import time
import os

def computation():
  time.sleep(2)
  return random.random()

def logic():
  try:
    res = computation()
    if res < 0.5:
      raise Exception('Low grade')
  except:
    traceback.print_exc()

if __name__ == "__main__":
  print os.getpid()
  while True:
    logic()
```

```
(pycon)root@theman:~/pycon# cat hook.py
import bytehook
import sys

hookid = bytehook.hook(sys.modules['__main__'].
logic, 6, with_state=True)
print hookid
```

pyrasite + bytehook

```
(pycon)root@theman:~/pycon# python test.py
11969
Traceback (most recent call last):
  File "test.py", line 14, in logic
    raise Exception('Low grade')
Exception: Low grade
Traceback (most recent call last):
  File "test.py", line 14, in logic
    raise Exception('Low grade')
Exception: Low grade
0
--Return--
> /root/pycon/local/lib/python2.7/site-packages/bytehook.py(159)runpdb()->None
-> pdb.set_trace()
(Pdb) a
_locals = {'res': 0.38844880564575357}
```

```
(pycon)root@theman:~/pycon# pyrasite 11969 hook.py
```

What is a .pyc file ?

An on disk bytecode representation to save compile time when reusing code.

[4 byte magic number] - Changes for different python versions, used to check that you are running it with a compatible python interpreter.

[4 byte timestamp] - Modification time of the original .py, used to check if the .pyc needs to be recreated because the original .py was modified.

[serialized code object] - Using the marshal protocol, encodes the code object that represents the module. Unlike pickle, this is CPython version dependent, and can only encode non-recursive primitive built-in python objects.

Some PYC gotchas

If you have a frozen environment you might want to consider using PYTHONDONTWRITEBYTECODE environment flag to disable creation of .pyc

The modified time resolution is seconds (at least in windows). If you have configuration .py files you override (mv config_a.py config.py), but both were created in the same second (let's say because you pulled them from version control), if a config.pyc already exists, python will still use it instead of recreating it.

(*) Python 3 also checks for the .py file size

Python 2 does not do I/O error checks when reading .py to compile a .pyc and thus may create valid .pyc that do not represent the original .py and keep using them till they are manually deleted (very rare, but happened to me a few times in the lab).

Some optimizations

```
>>> def add():
...     return 2 + 3

>>> def multiply_big():
...     return 'a' * 10000


>>> def opt2(a):
...     return 1 if not a == False else 2
```

```
>>> dis.dis(add)
  2           0 LOAD_CONST               3 (5)
              3 RETURN_VALUE
>>> dis.dis(multiply_big)
  2           0 LOAD_CONST               1 ('a')
              3 LOAD_CONST               2 (10000)
              6 BINARY_MULTIPLY
              7 RETURN_VALUE
>>> dis.dis(opt2)
  2           0 LOAD_FAST                0 (a)
              3 LOAD_GLOBAL              0 (False)
              6 COMPARE_OP               2 (==)
              9 POP_JUMP_IF_TRUE        16
             12 LOAD_CONST               1 (1)
             15 RETURN_VALUE
        >>   16 LOAD_CONST               2 (2)
             19 RETURN_VALUE
```

Questions ?

bytehook

bytehook is only interested in calling other functions from your code, so it’s relatively easy to read and figure out what it does, the main code (removed parts) is:

```
    code = func.func_code
    newconsts, noneindex, minusoneindex, hookpointindex = getoraddtotuple(code.co_consts, None, -1,
hookpointcounter)
    newnames, replaceindex, runhookpointindex = getoraddtotuple(code.co_names, __name__,
'run_hookpoint')
    newnames, localsindex, globalsindex = getoraddtotuple(newnames, 'locals', 'globals')
    pdbtracecode = createbytecode('LOAD_CONST', minusoneindex, 'LOAD_CONST', noneindex, 'IMPORT_NAME',
replaceindex, 'LOAD_ATTR', runhookpointindex, 'LOAD_CONST', hookpointindex, 'LOAD_GLOBAL', localsindex,
'CALL_FUNCTION', 0, 'LOAD_GLOBAL', globalsindex, 'CALL_FUNCTION', 0, 'CALL_FUNCTION', 3, 'POP_TOP')
    newcode = insertbytecode(code.co_code, 0, pdbtracecode)
    newlnotab = fixlines(code.co_lnotab, 0, len(pdbtracecode))
    newstacksize = code.co_stacksize + 4
    newfunc = new.code(code.co_argcount, code.co_nlocals, newstacksize, code.co_flags, newcode,
newconsts, newnames, code.co_varnames, code.co_filename, code.co_name, code.co_firstlineno, newlnotab,
code.co_freevars, code.co_cellvars)
    func.func_code = newfunc
```

bytehook’s magic

Use chr(dis.opmap[s]) to add the opcode to the bytecode stream and struct.pack ('H', short) to add the optional argument.

Since we are adding a self-contained byte-code into the (start/middle/end) of an existing one, we need to go over all of the jump opcodes (dis.hasjrel / dis.hasjabs) in the original byte-code and patch the jump targets relative to the correct position.

Also needed is to fix the mapping between the original source code lines and the new modified bytecode stream (remember co_lnotab?), a nice tuple of 2 byte string: addr_incr, line_incr in zip(co_lnotab[::2], co_lnotab[1::2]) where for each bytecode index increase, add a line number increase. We need to fix the line number offsets.

co_code

co_argcount	number of arguments (not including * or ** args)
co_code	string of raw compiled bytecode
co_consts	tuple of constants used in the bytecode
co_filename	name of file in which this code object was created
co_firstlineno	number of first line in Python source code
co_flags	bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg
co_lnotab	encoded mapping of line numbers to bytecode indices
co_name	name with which this code object was defined
co_names	tuple of names of local variables
co_nlocals	number of local variables (including arguments)
co_stacksize	virtual machine stack space required
co_varnames	tuple of names of arguments and local variables
co_cellvars	tuple of names of local variables that are referenced by nested functions
co_freevars	tuple containing the names of free variables

Bytecode stream

A python string that includes opcodes with optional argument.

1 unsigned char - the opcode.

2 unsigned char - an optional opcode argument (sometimes unsigned short).

There are about 100 bytecode, might be changed between python versions, you can tell the difference between those who have an argument and those that don't by looking at dis.HAVE_ARGUMENT (90 in 2.7.11).

```
>>> f.func_code.co_code
'|\x00\x00S'
>>> [ord(x) for x in f.func_code.co_code]
[124, 0, 0, 83]
```

```
>>> def f(a=1):
...     return a
...
```

Lib/dis.py utility

Python has a built in disassembler utility which can teach us:

Which objects have bytecode in them (_have_code)

What does a code object actually have (disassemble)

Which opcodes are jump targets (findlabels)

How does python map source code line numbers to bytecode (findlinestarts)

It’s manual page also lists all builtin opcodes, their arguments and what they do.

Main bytecode groups (not exhaustive)

Stack manipulation - POP_TOP, ROT_TWO, DUP_TOP

Operations - UNARY_NEGATIVE, BINARY_MULTIPLY, BINARY_SUBSCR

Function calls - CALL_FUNCTION, CALL_FUNCTION_VAR

Flow control - BREAK_LOOP, RETURN_VALUE, JUMP_FORWARD, JUMP_ABSOLUTE, POP_JUMP_IF_TRUE

Exception - RAISE_VARARGS, SETUP_EXCEPT

Load / Save - STORE_NAME, STORE_ATTR, LOAD_CONST

Print - PRINT_ITEM, PRINT_NEWLINE