implant Documentation

Oliver Berger

Jun 01, 2018

Contents

CHAPTER 1

Features

- Python >= 3.5 asyncio
- adhoc transferable remote procedures
- remote part of a *implant.core.Command* may reside in a separate module
- a *implant.core.Command* specific *implant.core.Channel* enables arbitrary protocols between local and remote side
- events
- quite small core
- tests

CHAPTER 2

Limitations

- Python >= 3.5
- only pure Python modules are supported for remote import, if no venv is used
- *implant.core.Command* s must reside in a module other then *__main__*
- at the moment sudo must not ask for password

CHAPTER 3

Example

3.1 General application

```
import asyncio
import pathlib

from implant import core, connect, commands


async def remote_tasks():
    # create a connector for a python process
    connector = connect.Lxd(
        container='zesty',
        hostname='localhost'
    )
    connector_args = {
        'python_bin': pathlib.Path('/usr/bin/python3')
    }
    # connect to a remote python process
    remote = await connector.launch(**connector_args)

    # start remote communication tasks
    com_remote = asyncio.ensure_future(remote.communicate())
    try:
        # execute command
        cmd = commands.SystemLoad()
        result = await remote.execute(cmd)

        print("Remote system load:", result)

    finally:
        # stop communication tasks
        com_remote.cancel()
        await com_remote
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(remote_tasks())
    loop.close()
```

3.2 An example Echo Command

```
import logging
import os

from implant import core


log = logging.getLogger(__name__)


class Echo(core.Command):

    """Demonstrate the basic command API."""

    async def local(self, context):
        """The local side of the RPC.

           :param context: :py:obj:`implant.core.DispatchLocalContext`
        """
        # custom protocol
        # first: send
        await context.channel.send_iteration("send to remote")

        # second: receive
        from_remote = []
        async for x in context.channel:
            from_remote.append(x)
        log.debug("*********** receiving from remote: %s", from_remote)

        # third: wait for remote to finish and return result
        remote_result = await context.remote_future

        result = {
            'from_remote': ''.join(from_remote),
        }
        result.update(remote_result)
        return result

    async def remote(self, context):
        """The remote side of the RPC.

           :param context: :py:obj:`implant.core.DispatchRemoteContext`
        """
        # first: receive
        from_local = []
        async for x in context.channel:
```

(continues on next page)

(continued from previous page)

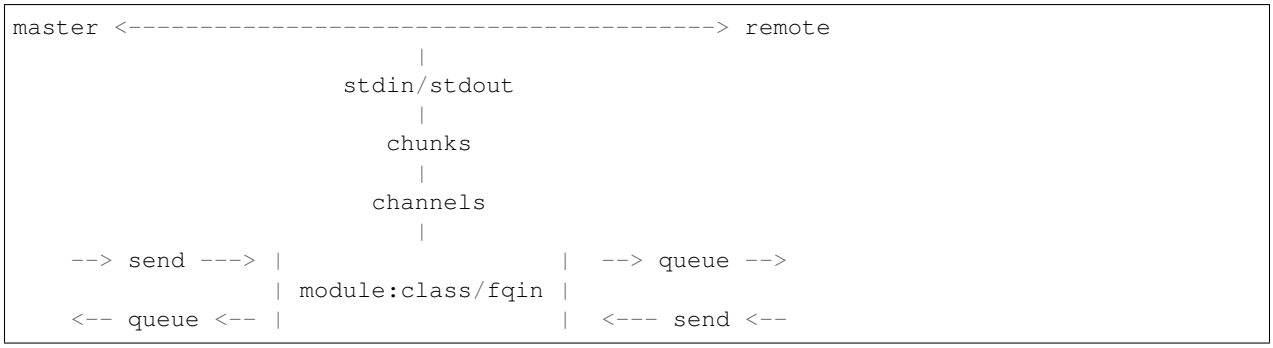
```
            from_local.append(x)
        log.debug("************ receiving from local: %s", from_local)

        # second: send
        await context.channel.send_iteration("send to local")

        # third: return result
        return {
            'from_local': ''.join(from_local),
            'remote_self': self,
            'pid': os.getpid()
        }
```

CHAPTER 4

Internals



CHAPTER 5

API

5.1 implant.core

The core module is transfered to the remote process and will bootstrap pipe communication.

It creates default channels and dispatches commands accordingly.

class implant.core.**BaseHeaderItem**(*, *default=None*, *encoder=None*, *decoder=None*)
Bases: object

A base item of a header.

static decode(*value*)
Decode the value of that item.

encode(*value*)
Encode the value of that item.

set_index(*index*)
Set the index of the item in the header.

class implant.core.**Channel**(*name=None*, *, *send*, *loop=None*)
Bases: asyncio.queues.Queue

Channel provides means to send and receive messages bound to a specific channel name.

__await__()
Receive the next message in this channel.

__init__(*name=None*, *, *send*, *loop=None*)
Initialize the channel.

Parameters

- **name** – the channel name
- **send** – the partial send method of Channels
- **loop** – the event loop

coroutine pop()
Get one item from the queue and remove it on return.

send = None
The send method bound to this channel's name. See *Channels.send()* for details.

coroutine send_iteration(*iterable*)
Send an iterable to the remote.

class implant.core.**Channels**(*reader*, *writer*, *, *loop=None*)
Bases: object

Hold references to all channel queues and route messages accordingly.

__init__(*reader*, *writer*, *, *loop=None*)
Create a *Channels* instance which delegates incomming messages into their appropriate *Channel* queues.

Parameters

- **reader** – asyncio.StreamReader
- **writer** – asyncio.StreamWriter
- **loop** – the event loop

coroutine _finalize_message(*buffer*, *chunk*)
Finalize the message if *Header.eom* is True.

This will also acknowledge the message if *Header.send_ack* is True.

coroutine _read_chunk()
Read a single chunk from the Channel.reader.

coroutine _receive_reader()
Start reception of messages.

coroutine _send_ack(*uid*)
Send an acknowledgement message.

Parameters **uid** – *Uid*

acknowledgements = None
Global acknowledgment futures distinctive by uid.

chunk_size = 32768

coroutine enqueue()
Schedule receive tasks.

Incomming chunks are collected and stored in the appropriate channel queue.

get_channel(*channel_name*)
Create a channel and weakly register its queue.

Parameters **channel_name** – the name of the channel to create

Returns *Channel* instance with a bound send method

incomming = None
A collection of all active channels.

log = <logging.Logger object>

coroutine send(*channel_name*, *data*, *ack=False*, *compress=6*)
Send data in a encoded form to the channel.

Parameters

- **channel_name** – the name of the channel
- **data** – the python object to send
- **ack** – request acknowledgement of the reception of that message
- **compress** – compress the data with zlib

Messages are split into chunks and put into the outgoing queue.

class implant.core.**Chunk**(*header*, *channel_name*, *data*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *header*, *channel_name*, *data*)
Create new instance of Chunk(header, channel_name, data)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new Chunk object from a sequence or iterable

_replace(***kwds*)
Return a new Chunk object replacing specified fields with new values

channel_name
Alias for field number 1

data
Alias for field number 2

header
Alias for field number 0

class implant.core.**Command**(***parameters*)
Bases: object

Common ancestor of all Commands.

command_name = 'implant.core:Command'

dispatch_data
Data to be dispatched.

parameters = {}

class implant.core.**CommandRemote**(*full_classname*)
Bases: object

Delegates remote task to another class.

This is usefull, if one wants not to import remote modules at the master side.

log = <logging.Logger object>

coroutine prepare()
Import the module for remote class.

set_remote_class(*module*)
Set remote class.

class implant.core.**ConnectionLostStreamReaderProtocol**(**args*, *connection_lost_cb*, ***kwargs*)
Bases: asyncio.streams.StreamReaderProtocol

Call a callback on connection_lost.

connection_lost(*exc*)
Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

class implant.core.**Core**(*loop*, ***, *echo=None*, ***kwargs*)
Bases: object

Core starts the *Dispatcher*.

coroutine communicate(*reader*, *writer*)
Start the dispatcher and register the *ShutdownRemoteEvent*.

On shutdown:

1. the import hook is removed
2. the *Dispatcher.dispatch* task is stopped
3. the *Channels.enqueue* task is stopped

coroutine connect(***, *stdin*, *stdout*, *stderr=None*)
Connect to stdin and stdout pipes.

coroutine connect_sysio()
Connect to sys.stdin and sys.stdout.

handle_connection_lost(*exc*)
We kill the process on connection lost, to avoid orphans.

log = <logging.Logger object>

classmethod main(*debug=False*, *log_config=None*, ***, *loop=None*, ***kwargs*)
Start the event loop and schedule core communication.

static setup_import_hook(*module_finder*)
Add module finder to sys.meta_path.

setup_logging(*debug=False*, *log_config=None*)
Setup a minimal logging configuration.

static teardown_import_hook(*module_finder*)
Remove a module finder from sys.meta_path.

class implant.core.**CustomEncoder**
Bases: object

Encode custom objects registered before.

class implant.core.**DispatchCommand**(*fqin*, *command_name*, *command_class*, *command_module*, *params*)
Bases: *implant.core.DispatchMessage*

Arguments for a command dispatch.

log = <logging.Logger object>

class implant.core.**DispatchException**(*fqin*, *exception*, *tb=None*)
Bases: *implant.core.DispatchMessage*

Remote execution ended in an exception.

implant.core.**DispatchLocalContext**
alias of implant.core.DispatchContext

class implant.core.**DispatchMessage**(*fqin*)
Bases: object

Base class for command dispatch communication.

coroutine __call__(*dispatcher*)
Executes appropriate *Dispatcher* methods to implement the core protocol.

log = <logging.Logger object>

class implant.core.**DispatchReady**(*fqin*)
Bases: *implant.core.DispatchMessage*

Set the dispatch ready.

implant.core.**DispatchRemoteContext**
alias of implant.core.DispatchContext

class implant.core.**DispatchResult**(*fqin*, *result=None*)
Bases: *implant.core.DispatchMessage*

The result of a remote execution.

class implant.core.**Dispatcher**(*channels*, *, *loop=None*)
Bases: object

Enables execution of *Command* s.

A *Command* is split into local and remote part, where a context with a dedicated *Channel* is provided to enable streaming of arbitrary data. The local part also gets a remote future passed, which resolves to the result of the remote part of the *Command*.

__init__(*channels*, *, *loop=None*)
Create a dispatcher, which executes messages on its own *Channel* to enable Command execution and communication via distinct *Channel* s.

coroutine _execute_channels()
Execute messages sent via our *Dispatcher.channel*.

channel = None
A channel for the dispatcher itself.

channels = None
The collection of all channels.

coroutine dispatch()
Start sending and receiving messages and executing them.

coroutine execute(*command_name*, **params*)
Execute a command.

First creating the remote side and its future and second executing its local part.

coroutine execute_dispatch_command(*fqin*, *command_name*, *params*)
Create a command and execute it.

coroutine execute_remote(*fqin*, *command*)
Execute the remote part of a *Command*. This method is called by a *DispatchCommand* message. The result is send via *Dispatcher.channel* to resolve the pending command future.

local_context(*fqin*, *remote_future*)
Create a local context to pass to a *Command* s local part.

The *Channel* is built via a fully qualified instance name (fqin).

log = <logging.Logger object>

pending_commands = None
Futures of *Command* s which are not finished yet.

pending_dispatches = None
A collection of dispatches, which are still not finished.

remote_context(*fqin*, *pending_remote_task*)
Create a remote context to pass to a *Command* s remote part.

The *Channel* is built via a fully qualified instance name (fqin).

remote_future(*fqin*, *command*)
Create a context for remote command future by sending *DispatchCommand* and returning its pending future.

set_dispatch_exception(*fqin*, *tb*, *exception*)
Set an exception for a pending command.

set_dispatch_ready(*fqin*)
Sets the pending dispatch ready, so the command execution continues.

set_dispatch_result(*fqin*, *result*)
Set a result for a pending command.

class implant.core.**ExceptionEncoder**
Bases: object

Encoder for Exception.

class implant.core.**FindSpecData**(***parameters*)
Bases: *implant.core.Command*

Find spec data for a module to import from the remote side.

command_name = 'implant.core:FindSpecData'

fullname
Define a *Command* parameter.

coroutine local(*context*)

parameters = {'fullname': <implant.core.Parameter object at 0x7fe751fd84e0>}

coroutine remote(*context*)

spec_data()
Find spec data.

class implant.core.**Flag**(*, *default=None*, *encoder=None*, *decoder=None*)
Bases: *implant.core.BaseHeaderItem*

A boolean flag of a header.

decode(*value*)
Decode the value of that item.

encode(*value*)
Encode the value of that item.

class implant.core.**Header**
Bases: bytes

The chunk header with flags and items.

channel_name_len
An item of a header.

compression
A boolean flag of a header.

data_len
An item of a header.

eom
A boolean flag of a header.

items = {'channel_name_len': <implant.core.HeaderItem object at 0x7fe751fcc278>, 'com

recv_ack
A boolean flag of a header.

send_ack
A boolean flag of a header.

size = 23

uid
An item of a header.

class implant.core.**HeaderItem**(*fmt*, ***kwargs*)
Bases: *implant.core.BaseHeaderItem*

An item of a header.

decode(*value*)
Decode the value of that item.

encode(*value*)
Encode the value of that item.

size
The size of the item.

class implant.core.**HeaderMeta**
Bases: type

Order items and set the size of the header.

classmethod apply_items_index(*items*)
Apply the index of each item.

class implant.core.**Incomming**(*, *connection_lost_cb=None*, *pipe=<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>*, *loop=None*)
Bases: asyncio.streams.StreamReader

A context for an incomming pipe.

coroutine connect()
Connect the pipe.

coroutine readexactly(*n*)
Read exactly n bytes from the stream.

This is a short and faster implementation then the original one (see of https://github.com/python/asyncio/issues/394).

class implant.core.**InvokeImport**(***parameters*)
Bases: *implant.core.Command*

Invoke an import of a module on the remote side.

The local side will import the module first. The remote side will trigger the remote import hook, which in turn will receive all missing modules from the local side.

The import is executed in a separate executor thread, to have a separate event loop available.

command_name = 'implant.core:InvokeImport'

fullname
Define a *Command* parameter.

coroutine local(*context*)

parameters = {'fullname': <implant.core.Parameter object at 0x7fe751fd84a8>}

coroutine remote(*context*)

class implant.core.**NoDefault**
Bases: object

Just a marker class to represent no default.

This is to separate really nothing and *None*.

class implant.core.**NotifyEvent**(***parameters*)
Bases: *implant.core.Command*

Notify about an event.

If the remote side registers for this event, it gets notified.

command_name = 'implant.core:NotifyEvent'

dispatch_local
Define a *Command* parameter.

event
Define a *Command* parameter.

coroutine local(*context*)

log = <logging.Logger object>

parameters = {'dispatch_local': <implant.core.Parameter object at 0x7fe751fd8470>, 'e

coroutine remote(*context*)

class implant.core.**OrderedMeta**
Bases: type

Preserve the order of instance creation.

items = [<implant.core.Flag object>, <implant.core.Flag object>, <implant.core.Flag ob

classmethod ordered_items(*dct*, *cls_order=None*)
Sort and filter items by type and instance creation.

class implant.core.**Outgoing**(*, *pipe=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *reader=None*, *loop=None*)
Bases: object

A context for an outgoing pipe.

coroutine connect()
Connect the pipe.

class implant.core.**Parameter**(*, *default=<class 'implant.core.NoDefault'>*, *description=None*)
Bases: object

Define a *Command* parameter.

exception implant.core.**RemoteClassNotSetException**
Bases: Exception

Raised when remote class is not set for *CommandRemote*

class implant.core.**RemoteModuleFinder**(*dispatcher*, *, *loop*)
Bases: importlib.abc.MetaPathFinder

Import hook that execute a *FindSpecData* command in the main loop.

See pep-0302, pep-0420 and pep-0451 for internals.

find_spec(*fullname*, *path*, *target=None*)
Find the spec of the module.

log = <logging.Logger object>

class implant.core.**RemoteModuleLoader**(*source*, *filename=None*, *is_package=False*)
Bases: importlib.abc.ExecutionLoader

Load the found module spec.

get_filename(*fullname*)
Abstract method which should return the value that __file__ is to be set to.

Raises ImportError if the module cannot be found.

get_source(*fullname*)
Abstract method which should return the source code for the module. The fullname is a str. Returns a str.

Raises ImportError if the module cannot be found.

is_package()
Optional method which when implemented should return whether the module is a package. The fullname is a str. Returns a bool.

Raises ImportError if the module cannot be found.

classmethod module_repr(*module*)
Return a module's repr.

Used by the module type when the method does not raise NotImplementedError.

This method is deprecated.

class implant.core.**SetEncoder**
Bases: object

Encoder for set.

class implant.core.**ShutdownRemoteEvent**
Bases: object

A Shutdown event.

Shutting down a remote connection is done by gracefully canceling all remote tasks. See *Core.communicate* for details.

class implant.core.**StopAsyncIterationEncoder**
Bases: object

Encoder for StopAsyncIteration.

class implant.core.**TupleEncoder**
Bases: object

Encoder for tuple.

class implant.core.**Uid**(*bytes=None*)
Bases: uuid.UUID

A unique id, which is basically a uuid.uuid1 instance.

time
The timestamp of the uuid1.

coroutine implant.core.**async_import**(*fullname*, *, *loop=None*)
Import module via executor.

coroutine implant.core.**event_dispatch**(*event*)
Dispatch an event to every handler.

implant.core.**event_handler**(*event_class*, *handler_=None*, *decorator=False*)
Define an event handler for a (new-style) class.

This can be called with a class and a handler, or with just a class and the result used as a handler decorator.

implant.core.**exclusive**(*fun*)
Make an async function call exclusive.

coroutine implant.core.**notify_event**(*event*)
Notify all subscribers of event.

class implant.core.**reify**(*wrapped*)
Bases: object

Taken from pyramid: create a cached property.

implant.core.**split_data**(*data*, *size=1024*)
Create a generator to split data into chunks.

5.2 implant.bootstrap

Bootstrap of a remote python process.

class implant.bootstrap.**Bootstrap**(*code*, *options=None*)
Bases: dict

Provide an iterator over the bootstrap code.

formatsourcelines(*lines*)
Remove full line comments.

5.3 implant.connect

Remote connection is established by a *Connector*.

class implant.connect.**Connector**
Bases: object

Base Connector class.

class implant.connect.**ConnectorMeta**
Bases: abc.ABCMeta

Connector meta base.

base
alias of *Connector*

connectors = {'local': <class 'implant.connect.Local'>, 'lxd': <class 'implant.conne

scheme
The unique connector scheme is the lowered class name.

class implant.connect.**ConnectorParams**
Bases: *implant.connect.ConnectorParams*

classmethod create(*connector*)

create_connector()
Lookup the connector for that string.

classmethod parse(*connection_str*)
Parse the connection string into its parts.

unparse()

class implant.connect.**Local**(*, *sudo=None*)
Bases: *implant.connect.SubprocessConnector*

A *Connector* to a local python process.

arguments(*, *code=None*, *options=None*, *python_bin=None*)
Iterate over the arguments to start a process.

Parameters

- **code** – the code to bootstrap the remote process
- **options** – options for the remote process
- **python_bin** – the path to the python binary

sudo

class implant.connect.**Lxd**(*, *container*, *hostname=None*, *user=None*, *sudo=None*)
Bases: *implant.connect.Ssh*

A *Connector* for accessing a lxd container.

If the hostname is omitted, the lxd container is local.

arguments(*, *code=None*, *options=None*, *python_bin=None*)
Iterate over the arguments to start a process.

Parameters

- **code** – the code to bootstrap the remote process

- **options** – options for the remote process
- **python_bin** – the path to the python binary

container

hostname

sudo

user

class implant.connect.**Remote**(*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*)
Bases: object

A remote receiving commands.

coroutine communicate()
Schedule the dispatcher.

coroutine execute(**args*, ***kwargs*)
Just call dispatcher.execute.

coroutine wait()
Wait for Remote to finish.

exception implant.connect.**RemoteMisbehavesError**
Bases: Exception

Exception is raised, when a remote process seems to be not what we expect.

class implant.connect.**Ssh**(*, *hostname=None*, *user=None*, *sudo=None*)
Bases: *implant.connect.Local*

A *Connector* for remote hosts reachable via SSH.

If a hostname is omitted, this connector acts like *Local*.

arguments(*, *code=None*, *options=None*, *python_bin=None*)
Iterate over the arguments to start a process.

Parameters

- **code** – the code to bootstrap the remote process
- **options** – options for the remote process
- **python_bin** – the path to the python binary

hostname

sudo

user

class implant.connect.**SubprocessConnector**
Bases: *implant.connect.Connector*

A *Connector* uniquely defines a remote target.

arguments(*, *code=None*, *options=None*, *python_bin=None*)
Iterate over the arguments to start a process.

Parameters

- **code** – the code to bootstrap the remote process
- **options** – options for the remote process

- **python_bin** – the path to the python binary

static bootstrap_code(*code=<module 'implant.core' from '/home/docs/checkouts/readthedocs.org/user_builds/implant/packages/implant-0.1.2-py3.5.egg/implant/core.py'>*, *options=None*)
Create the python bootstrap code.

coroutine launch(*, *loop=None*, *code=None*, *options=None*, *python_bin=None*, ***kwargs*)
Launch a remote process.

Parameters

- **code** – the python module to bootstrap
- **options** – options to send to remote
- **python_bin** – the path to the python binary to execute
- **kwargs** – further arguments to create the process

class implant.connect.**SubprocessRemote**(*transport*, *protocol*, *, *loop=None*)
Bases: *implant.connect.Remote*

A remote process.

kill()
Kill the process.

returncode
The exit code of the process.

send_signal(*signal*)
Send a sgnal to the process.

terminate()
Terminate the process.

coroutine wait()
Wait until the process exit and return the process return code.

coroutine implant.connect.**create_subprocess_remote**(*program*, **args*, *loop=None*, *limit=65536*, ***kwds*)
Create a remote subprocess.

5.4 implant.pool

5.5 implant.scripts

5.6 implant.testing

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W