wishpy

Release 0.0.9

Sep 21, 2020

Contents:

‘wishpy’ implements Python bindings for Wireshark, a versatile packet analysis tool and libpcap a widely used library for network packet capturing.

Main goal of the ‘wishpy’ project is to bring the wireshark’s dissectors to the Python world and make them available to the rich ecosystem of Python’s data analysis and visualization tools.

CHAPTER 1

Introduction

Python Bindings for `Wireshark` and `libpcap`

1.1 What `wishpy` is?

- Uses `cffi` to generate Python bindings for `wireshark` and `libpcap`
- You can write applications like `tshark` in Python
- Makes wireshark's dissectors available in Python and makes `libpcap` easily available in Python for packet capture
- This is in active development, but should work on common Linux distributions, if it doesn't please file an issue.
- Also,a drop-in replacement for pcapy. Supports all the major `pcapy` APIs.
- Early Windows support. Please check README-windows .

1.2 Getting Started

This packages right now is tested only on Linux (specifically Ubuntu 16.04) To be able to get started, following development environment is required -

1. `gcc` and it's toolset
2. Python 3.5 or higher and Python development environment.
3. Supports PyPy 7.3 or higher (compatible with Python 3.6)

It is highly recommended to start with a virtual environment, something like `virtualenv venv`

Typically simply doing a `python setup.py install` should be enough to get you started. If everything goes well, one will have the modules installed in the `site-packages`.

Once the packages are installed, you can run the example code -

Alternatively, if you just want to use wrapped APIs, they are used in - 1. `wishpy/scripts/tcpdump.py <interface_name>` (For live capturing the packets and dumping `json`, **NOTE:** Requires `sudo` permissions.) 2. `wishpy/scripts/tshark.py <pcap-file-path>` (For dumping packets from a `pcap``ish file as ``json`)

1.3 Wireshark support

Right now both Wireshark 2.6.x and wireshark 3.2.x are supported.

The best way to make sure this works is through `pkg-config`. Right now, default support is for `wireshark` 2.6 that ships with Ubuntu. If you have both the versions installed, it's a little bit tricky. If building `wireshark` from source, If you perform a `make install` (or `sudo make install`), the right `wireshark.pc` file is created and will be used during build.

1.4 `libpcap` support

libpcap library > 1.7 is supported. Also, there is a `pcapy` module, that can be used as a drop in replacement for pcapy. Similar APIs as `pcapy` are supported. We have performed quick testing with following versions of `libpcap` on Ubuntu (based on git tag) - `libpcap-1.7.4`, `libpcap-1.8.1`, `libpcap-1.9.1`.

1.5 Documentation

We have started with some very 'basic' Dissector/Capturer API. See `wishpy/scripts/tshark.py` to see how it can be used. This API is very early (in fact this is not really an API, but just a hint about what API might look like.) and very likely to change going forward. A very early version of the API Documentation is available.

1.6 Examples

See the code in `wishpy/scripts/` directory for how to use wishpy API.

A More detailed example using `wishpy` for publishing to Redis is available at the following repo -

- wishpy-examples

CHAPTER 2

API

2.1 Wireshark Dissector APIs

APIs for wireshark's dissectors.

This module provides consistent APIs for using wireshark's dissector in different scenarios. viz. using with live packet capture and using with a PCAP file. A couple of dissector classes are provided that can be directly used.

WishpyDissectorQueuePython : Can be used with *wishpy.libpcap.lib.capturer.WishpyCapturer* *WishpyDissectorFile*: Can be used for printing json data from a pcap(ish) file.

Example:

```
>>> d = WishpyDissectorFile('file.pcap')
>>> for packet in d.run():
    print(packet)
```

class wishpy.wireshark.lib.dissector.**WishpyDissectorBase**(**args*, ***kw*)

A Class that wraps the underlying dissector from epan module of `libwireshark`. Right now this simply prints the dissector tree.

__init__(**args*, ***kw*)

Initialize self. See help(type(self)) for accurate signature.

apply_filter(*filter_str*, *overwrite=False*)

Applies a filter given by the filter_str to the dissection.

Args: filter_str: str - A string that is in a wireshark filter format.

Returns: result: (int, str) - Result of application 0 success. Negative value suggesting error. Caller should check the error.

Note: Right now it is recommended to run this method before *run* method is called on the dissector.

cleanup_epan_dissector()

Cleans up internal dissector object.

This method should be called as the last part of *run()* method.

clear_filter()
Clears the dfilter if any.

classmethod enable_json_test()
Enable `json.loads` test of the generated Json.

init_epan_dissector()
Initializes `epan_dissect_t` and `epan_session` objects. These objects are passed to the *run* method.

classmethod packet_to_json(*handle_ptr*)
An example method that depicts how to use internal dissector API.

classmethod print_dissected_tree_details(*dissector*)
Packet Details view of the Dissected Tree like Wireshark Packet Details.

classmethod print_dissected_tree_details_api(*dissector*)
Print a packets Protocol tree using *proto_tree_json* API

(Note: This method should not be used, instead use *print_dissected_tree_details*.)

classmethod print_dissected_tree_details_node(*node_ptr*, *level*)
Print details of a single Node.

(Note: This method should not be called directly as yet, instead, one should call the *print_dissected_tree_details* method, which internally calls this.)

classmethod print_dissected_tree_json_node(*node_ptr*, *level=1*)
Returns a string representing dissected tree using the *ftypes* API.

classmethod print_dissected_tree_json_node_pretty(*node_ptr*, *level=1*)
Returns a string that represents a dissected tree.

classmethod print_dissected_tree_json_pretty(*dissector*)
Pretty prints dissected tree.

run(**args*, ***kw*)
A generator function `yield` ing at-least the dissected packets.

Implementing this as a generator function helps one to run code that looks like

```
>>> for dissected in dissector.run(count=1):
    # do stuff with the dissected packet
```

This is particularly convenient while performing live capture on an interface or dissecting a huge file.

classmethod set_elasticky(*enabled*)
Enable Elastic Compatible Json output.

classmethod set_pretty_print_details(*enabled=False*, *add_proto_tree=False*)
Set Pretty Printing and Details of a Packet Field.

class wishpy.wireshark.lib.dissector.**WishpyDissectorFile**(*filename*)
Dissector class for PCAP Files.

__init__(*filename*)
Initialize self. See help(type(self)) for accurate signature.

run(*count=0*, *skip=-1*)
Actual function that performs the Dissection.

Args: count (int, optional): The number of packets to run for. skip (int, optional): Skip the number of packets.

Raises: *WishpyErrorWthOpen*: If the handle cannot be opened.

Right now since we are only supporting dissecting packets from Wiretap supported files, only dissects packets from a pcap(ish) file.

class `wishpy.wireshark.lib.dissector.`**`WishpyDissectorQueue`**`(`**args*, ***kw*`)`
Dissector class for packets received from a Queue(ish) object.

`dissect_one_packet``()`
Dissects a single packet

Calls the *`fetch()`* method to fetch a single packet and then performs dissection using the wrapped `epan_perform_one_packet_dissection`.

`fetch``()`
Implement this function to fetch a single packet from the queue.

Implementation of this function should return the object of the type *Hdr* and *PacketData*

class `wishpy.wireshark.lib.dissector.`**`WishpyDissectorQueuePython`**`(`*queue*, *iface_name=None*`)`
Dissector class for Python Standard Library Queue.

`__init__``(`*queue*, *iface_name=None*`)`
Constructor

Args: queue: A python Queue like object

iface_name (str, optional): Name of the interface

`fetch``()`
Blocking Fetch from a Python Queue.

Returns: (hdr, data): A tuple containing PCAP like header and packet data.

`run``(`*count=0*`)`
Runs the dissection function for the packets fetched from the queue.

Args: count (int, optional): The number of packets to run dissector for.

Yields: packet: A dissected json of the packet.

if count is <= 0, infinite iterator. This iterator can be stopped by receiving an object of the form (stop, None) from the queue. So the write of the queue will have to ensure this. Or Call the *stop* method.

`stop``()`
Stop's the generator by setting internal state.

exception `wishpy.wireshark.lib.dissector.`**`WishpyEpanLibAlreadyInitialized`**
Error raised trying to initialize already initialized EPAN Library.

exception `wishpy.wireshark.lib.dissector.`**`WishpyEpanLibUninitializedError`**
Error raised during initialization of EPAN library.

exception `wishpy.wireshark.lib.dissector.`**`WishpyErrorInitDissector`**
Error raised during initialization of dissector and or dissector session.

exception `wishpy.wireshark.lib.dissector.`**`WishpyErrorWthOpen`**
Error raised during opening a Pcap file.

`wishpy.wireshark.lib.dissector.`**`cleanup_process`**`()`
Per process cleanup. de-init of epan/wtap modules.

`wishpy.wireshark.lib.dissector.`**`setup_process`**`()`
Per process initialization.

This method should be called once per process (note: Not thread.) This will perform underlying library initialization, so that eventually dissectors can *run*.

2.2 `libpcap` Capturer APIs

Capture API using the libpcap.

class `wishpy.libpcap.lib.capturer.`**`PCAPHeader`**(*dltype*, *ts_sec*, *ts_usec*, *len*, *caplen*)

`caplen`
captured length of the packet.

`dltype`
Data Link Layer type as per libpcap

`len`
length of the packet.

`ts_sec`
seconds part of the timestamp.

`ts_usec`
micro-seconds part of the timestamp.

class `wishpy.libpcap.lib.capturer.`**`WishpyCapturer`**
Base WishpyCapturer class.

Following API are provided *open*, *close*, *start*, *stop*.

`close`()
close: Perform any resource cleanup related to the capturer.

`open`()
open: Opens the capturer.

Use this to perform any Capturer specific initialization. For instance, our API deals with Packet Capture Pipelines that can be connected using Python Queues. So Setting up Queues etc. can be performed here in this method.

`start`(***kw*)
start: Start actual capture of packets.

Implement in such a way tha it should be possible to start/stop capture multiple times for an instance of the capturer.

`stop`(***kw*)
stop: Stop actual capture of packets.

Implement in such a way that it should be possible to start/stop capture multiple times for an instance of the capturer.

exception `wishpy.libpcap.lib.capturer.`**`WishpyCapturerCaptureError`**

class `wishpy.libpcap.lib.capturer.`**`WishpyCapturerFileToQueue`**(*filename*, *queue*, ***kw*)
A Libpcap capturer class that wraps a PCAP file.

This class provides the *Capturer* API wrapping a PCAP file. Note: For the dissection part it is better to directly use *`wishpy.wireshark.lib.dissector.WishpyDissectorFile`*. This class should be used when you want to take packets from a PCAP file and do something other than ‘dissect’ing them.

`__init__`(*filename*, *queue*, ***kw*)
Constructor

Args: filename: PCAP file to be opened for reading.

queue: Queue to send packets to.

`close``()`
Closes internal *libpcap* handle

libpcap's *pcap_close* function is called and our activated flag is set to False.

`open``()`
Opens the filename for PCAP Capture.

Returns: None Raises: WishpyCapturerOpenError: If failure to open a file.

`class` `wishpy.libpcap.lib.capturer.`**`WishpyCapturerIfaceToQueue`**`(`*iface*, *queue*, *snaplen=0*, *promisc=True*, *timeout=10*, ***kw*`)`
`libpcap` based packet capturer for an interface on the system.

This capturer captures packet from the OS interface and posts them, on the Queue. Right now it is not completely abstracted out what gets posted on the queue. Assume they are tuples like - (header, data)

`__init__``(`*iface*, *queue*, *snaplen=0*, *promisc=True*, *timeout=10*, ***kw*`)`
Constructor

Args: queue: Python Queue like objects that supported Get/Put APIs.

Get/Put APIs should be thread/process safe. (eg. Queue, multiprocessing Queue etc.)

iface: string - An Interface Name on the local OS.

snaplen: integer (optional), Maximum Capture length of the data.

Default - Don't set Capture length (ie. if input value is 0.)

promisc: Boolean (optional). Default True To determine whether to start capturing in 'promiscuous' mode.

timeout: integer - Timeout in miliseconds to wait before next 'batch' of captured packets is returned

(This value maps directly to *libpcap: packet buffer timeout*.) Default value is 10ms. Use lower values for more 'responsive' capture, higher values for larger batches.

**kw: Possible Keyword argument's that can be supported.

`close``()`
Closes internal *libpcap* handle

libpcap's *pcap_close* function is called and our activated flag is set to False.

`open``()`
Open's the Capturerer readying it for performing capture.

Calls libpcap's *pcap_create* and depending upon requested parameters during the Constructor, those values are set and finally activates the handle.

`exception` `wishpy.libpcap.lib.capturer.`**`WishpyCapturerOpenError`**

`class` `wishpy.libpcap.lib.capturer.`**`WishpyCapturerQueue`**`(`*queue*, ***kw*`)`
Base Class for Sending Packets to Python Queue like objects.

`__init__``(`*queue*, ***kw*`)`
Constructor

Args: queue: Python Queue like objects that supported Get/Put APIs

The Get/Put APIs should be in a thread/process safe manner. (eg. Queue, multiprocessing Queue etc.)

**kw: Possible keyword arguments.

`start`(*count=-1*, *serialize=True*)

Starts capturing of the packets.

Note: This is a blocking function.

An application should call this function from a separate thread of execution. Calls internal `pcap_loop` function of `libpcap`.

Args:

count: (optional) if specified should be a positive integer specifying maximum number of packets to be captured.

serialize: (optional) bool - Serialize data If specified serializes the header and data to `PCAPHeader` and `bytes` objects (default `True`)

Returns: On Success Nothing.

When `pcap_loop` returns, A special tuple - `('stop', b'')` is placed on the queue. For the consumer of the queue, this should signal end of data transfer from the producer.

Raises: On Error Condition *wishpy.wireshark.lib.WishpyCapturerCaptureError*.

`stop`()

Stops the capture.

Simply calls internal `libpcap`'s `pcap_breakloop`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index