Workbench Documentation

Release 0.1

Brian Wylie

July 21, 2014

Contents

A medium-data framework for security research and development teams.

Workbench focuses on simplicity, transparency, and easy on-site customization. As an open source python project it provides light-weight task management, execution and pipelining for a loosely-coupled set of python classes.

CHAPTER 1

Email Lists (Forums)

- Users Email List: Users_Email_List
- Developers Email List: Developers_Email_List

1.1 Contents

1.1.1 Detailed Project Description

The workbench project takes the workbench metaphore seriously. It's a platform that allows you to do work; it provides a flat work surface that supports your ability to combine tools (python modules) together. In general a workbench never constrains you (oh no! you can't use those 3 tools together!) on the flip side it doesn't hold your hand either. Using the workbench software is a bit like using a Lego set, you can put the pieces together however you want AND adding your own pieces is super easy!.

Loosely coupled

- No inheritance relationships
- No knowledge of data structures
- Just take some input and barf some output (no format requirements)

Flat

- Workers (that's it... everything is a worker)
- Server dynamically loads workers from a directory called 'workers'

Robust

- Worker fails to load (that's fine)
- Worker crashes (no sweat, that request fails but system chugs on)

Transparency

- All worker output is reflected in the data store (currently Mongo)
- Use RoboMongo (see below) to inspect exactly what workers are outputting.

Small Granularity

- The system works by passing references from one worker to another so there is NO benefit to large granularity workers.
- It's super easy to have a worker that aggregates information from a set of workers, the opposite (breaking apart a large code chunk into smaller units) is almost never easy.
- Pull just what you want, workers and views (which are just workers) can be selectve about exactly which fields get pulled from which workers.

1.1.2 Installing Workbench

Workbench Server (Minimum Install)

The workbench server is extremely robust to worker failure. In fact it can run without many of the dependencies so you can setup a server quickly with 'Minimum Install' and than later do a 'Full Install'.

Mac/OSX

```
$ brew install mongodb
```

Ubuntu (14.04 and 12.04)

```
$ sudo apt-get install mongodb
$ sudo apt-get install python-dev
$ sudo apt-get install g++
```

Workbench Python Modules

```
$ pip install workbench --pre
$ workbench_server
```

That's it, the workbench server will come up and is ready to start servicing requests. Note: Some workers will fail to load but that is fine, to have all workers run see 'Full Install'.

Workbench Client(s)

```
$ pip install workbench --pre
$ workbench (this runs the Workbench CLI)
```

That's it!

If you have a workbench server setup (somewhere) you can now start the workbench CLI client, or any of the existing clients (in workbench/clients) or even start writing your own clients against that server (see *Making your own Client*)

Workbench Server (Full Install)

Mac/OSX

```
$ brew install mongodb
$ brew install yara
$ brew install libmagic
$ brew install bro
```

Important: Put the bro executable in your PATH (/usr/local/bin or wherever bro is)

Ubuntu (14.04 and 12.04)

```
$ sudo apt-get install mongodb
$ sudo apt-get install python-dev
$ sudo apt-get install g++
$ sudo apt-get install libssl0.9.8
```

- **Bro IDS: In general the Bro debian package files are WAY too locked down with dependencies on exact versions of li**
  - sudo dpkg -i Bro-2.2-Linux-x86_64_flex.deb
  - **If using the Debian package above doesn't work out:**
    - Check out the Installation tutorial bro_install
    - or this one bro_starting
    - or go to offical Bro Downloads www.bro.org/download/

Important: Put the bro executable in your PATH (/opt/bro/bin or wherever bro is)

Install Indexers

The indexers 'Neo4j' and 'ElasticSearch' are optional. We strongly suggest you install both of them but we also appreciate that there are cases where that's not possible or feasible.

Mac/OSX

```
$ brew install elasticsearch
$ pip install -U elasticsearch
$ brew install neo4j
```

- Note: You may need to install Java JDK 1.7 Oracle JDK 1.7 DMG for macs.

Ubuntu (14.04 and 12.04)

- Neo4j: See official instructions for Neo4j here
- Note: You may need to install Java JDK 1.7. If you have Java 1.7 installed and error says otherwise, run

```
$ update-alternatives --config java and select Java 1.7
```

- ElasticSearch:
  - wget https://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-1.2.1.deb
  - sudo dpkg -i elasticsearch-1.2.1.deb
  - sudo update-rc.d elasticsearch defaults 95 10
  - sudo /etc/init.d/elasticsearch start
  - Any issues see elasticsearch_webpage

Workbench Python Modules

Note: Workbench is continuously tested with python 2.7. We're currently working on Python 3 support (Issue 92).

For quick spinup just pull Workbench down from pip. If you're going to do development

```
$ pip install workbench --pre
$ workbench_server
```

OR

```
$ cd workbench
$ python setup.py develop
$ workbench_server
```

Optional Tools

Robomongo

Robomongo is a shell-centric cross-platform MongoDB management tool. Simply, it is a handy GUI to inspect your mongodb.

- http://robomongo.org/
- download and follow install instructions
- create a new connection to localhost (default settings fine). Name it as you wish.

Dependency Installation Errors

Python Modules

Note: If you get a bunch of clang errors about unknown arguments or 'cannot link a simple C program' add the following FLAGs:

```
$ export CFLAGS=-Qunused-arguments
$ export CPPFLAGS=-Qunused-arguments

**Errors when running Tests**
```

If when running the worker tests you get some errors like 'MagicError: regexec error 17, (illegal byte sequence)' it's an issue with libmagic 5.17, revert to libmagic 5.16. Using brew on Mac:

```
$ cd /usr/local
$ brew versions libmagic # Copy the line for version 5.16, then paste (for me it looked like the
$ git checkout bfb6589 Library/Formula/libmagic.rb
$ brew uninstall libmagic
$ brew install libmagic
```

1.1.3 Running WorkBench

Server (localhost or server machine)

```
$ pip install workbench --pre
$ workbench_server
```

CLI (Command Line Interface)

:: $ workbench

Example Clients (use -s for remote server)

There are about a dozen example clients showing how to use workbench on pcaps, PEfiles, pdfs, and log files. We even have a simple nodes.js client (looking for node devs to pop some pull requests :).

```
$ cd workbench/clients
$ python simple_workbench_client.py [-s tcp://mega.server.com]
```

Workbench Examples

- PCAP to Graph (A short teaser)
- Workbench Demo
- Adding a new Worker (super hawt)
- PCAP to Dataframe
- PCAP DriveBy Analysis
- Using Neo4j for PE File Sim Graph
- Generator Pipelines Notebook
- WIP Notebooks
  - Network Stream Analysis Notebook
  - PE File Static Analysis Notebook

Making your own Worker

- See the notebook Adding a new Worker

Workbench Conventions

Workers should adhere to the following naming conventions (not enforced)

- **If you work on a specific type of sample than start the name with** that
- Examples: pcap_bro.py, pe_features.py, log_meta.py
- **A worker that is new/experimental should start with 'x_'** (x_pcap_razor.py)
- **A 'view'(worker that handles 'presentation') should start with** 'view_'
- Examples: view_log_meta.py, view_pdf.py, view_pe.py

Making your own Client

Although the Workbench repository has dozens of clients (see workbench/clients)there is NO official client to workbench. Clients are examples of how YOU can just use ZeroRPC from the Python, Node.js, or CLI interfaces. See ZeroRPC.

```
import zerorpc
c = zerorpc.Client()
c.connect("tcp://127.0.0.1:4242")
with open('evil.pcap','rb') as f:
    md5 = c.store_sample('evil.pcap', f.read())
print c.work_request('pcap_meta', md5)
```

Output from above 'client':

```
{'pcap_meta': {
    'encoding': 'binary',
    'file_size': 54339570,
    'file_type': 'tcpdump (little-endian) - version 2.4 (Ethernet, 65535)',
    'filename': 'evil.pcap',
    'import_time': '2014-02-08T22:15:50.282000Z',
    'md5': 'bba97e16d7f92240196dc0caef9c457a',
    'mime_type': 'application/vnd.tcpdump.pcap'
}}``
```

Running the IPython Notebooks

```
brew install freetype
brew install gfortran
pip install -r requirements\_notebooks.txt
Go to Starbucks..
```

Running Tests

Unit testing, sub-pipeline tests, and full pipeline tests

```
$ tox
```

Benign Error

We have no idea why occasionaly you see this pop up in the server output. To our knowledge it literally has no impact on any functionality or robustness. If you know anything about this please help us out by opening an issue and pull request. :)

```
ERROR:zerorpc.channel:zerorpc.ChannelMultiplexer, unable to route event:
_zpc_more {'response_to': '67d7df3f-1f3e-45f4-b2e6-352260fa1507', 'zmqid':
['\x00\x82*\x01\xea'], 'message_id': '67d7df42-1f3e-45f4-b2e6-352260fa1507',
'v': 3} [...]
```

VirusTotal Warning

The vt_query.py worker uses a shared 'low-volume' API key provided by SuperCowPowers LLC. When running the vt_query worker the following warning happens quite often:

```
"VirusTotal Query Error, no valid response... past per min quota?"
```

If you'd like to use the vt_query worker on a regular basis, you'll have to put your own VirusTotal API key in the workbench/server/config.ini file.

Configuration File Information

When you first run workbench it copies default.ini to config.ini within the workbench/server directory, you can make local changes to this file without worrying about it getting overwritten on the next 'git pull'. Also you can store API keys in it because it never gets pushed back to the repository.

```
# Example/default configuration for the workbench server
[workbench]

# Server URI (server machine ip or name)
# Example: mybigserver or 12.34.56.789
server_uri = localhost

# DataStore URI (datastore machine ip or name)
# Example: mybigserver or 12.34.56.789
datastore_uri = localhost

# Neo4j URI (Neo4j Graph DB machine ip or name)
# Example: mybigserver or 12.34.56.789
neo4j_uri = localhost

# ElasticSearch URI (ELS machine ip or name)
# Example: mybigserver or 12.34.56.789
els_uri = localhost

# DataStore Database
# Example: customer123, ml_talk, pdf_deep
database = workbench

# Storage Limits (in MegaBytes, 0 for no limit)
worker_cap = 10
samples_cap = 200

# VT API Key
```

```
# Example: 93748163412341234v123947
vt_apikey = 123
```

1.1.4 Frequently Asked Questions

Medium Data

- **What do you mean by 'medium' data?** The developers of Workbench feel like Medium-Data is a sweet spot, large enough to be meaningful for model generation, statistics and predictive performance but small enough to allow for low latency, fast interaction and streaming 'hyperslabs' from server to client.
- **What do you mean by hyperslabs?** Many of our examples (notebooks) illustrate the streaming generator chains that allow a client (python script, IPython notebook, Node.js, CLI) to stream a filtered subset of the data over to the client.
- **Why do you have exploding heads every time you talk about streaming data into a DataFrame?** Once you efficiently (streaming with zero-copy) populate a Pandas dataframe you have access to a very large set of statistics, analysis, and machine learning Python modules (statsmodel, Pandas, Scikit-Learn).
- **What kind of hardware do you recommend for the Workbench server?** Workbench server will run great on a laptop but when you're working with a group of researchers the most effective model is a shared group server. A beefy Dell server with 192Gig of Memory and a 100 TeraByte disk array will allow the workbench server to effectively process in the neighborhood of a million samples (PE Files, PDFs, PCAPs, SWF, etc.)

Client/Server

- **Philosophy on local Workbench server.** As you've noticed from many of the documents and notebooks, Workbench often defaults to using a local server. There are several reasons for this approach:
  - We love the concept of git, with a local server (sandbox) for quickness and agility and a remote server for when your ready to share your changes with the world.
  - Workbench embraces this approach: Developers can quickly develop new fuctionality on their local server and when they are ready to share the awesome they can 'push' their new worker to the 'group server'.
- **How do I push my worker to a 'group server'?**
  - development box: $ git push
  - server box: $ git pull
- **How do I have my workbench clients hit a remote server?**
  - All clients have a -s, –server argument:

    ```
    $ python pcap_bro_indexer.py   # Hit local server
    $ python pcap_bro_indexer.py -s = my_server  # Hit remote server
    ```

  - **All clients read from the config.ini in the clients directory** If you always hit a remote server simply change the config.ini in the clients directory to point to the groupserver.:

    ```
    server_uri = localhost  (change this to whatever)
    ```

- Okay I've changed my config.ini file, and now it shows up when I do a '$ git status'. How do I have git ignore it?:

```
git update-index --assume-unchanged workbench/clients/config.ini
git update-index --assume-unchanged workbench/server/config.ini
```

- **How do I setup a development server and a production server?** In general workbench should be treated like any other python module and it shouldn't add any complexity to existing development/QA/deployment models. One suggestion (to be taken with a grain of salt) is simply to use git braches.:

  ```
  $ git checkout develop (on develop server)
  $ git checkout master (on prod server)
  ```

Cow Points

- Are Cow Points worth anything? : No
- Will Cow Points ever be worth anything? : Maybe
- Are Cow Points officially tracked? : Yes
- Will I receive good Karma for Cow Points? : Yes

1.1.5 Contributing

Report a Bug or Make a Feature Request

Please go to the GitHub Issues page: https://github.com/SuperCowPowers/workbench/issues.

Look at the Code

> **Warning:** Caution!: The repository contains malcious data samples, be careful, exclude the workbench directory from AV, etc...

```
git clone https://github.com/supercowpowers/workbench.git
```

Become a Developer

Workbench uses the 'GitHub Flow' model: GitHub Flow

- To work on something new, create a descriptively named branch off of master (ie: my-awesome)
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a pull request
- After someone else has reviewed and signed off on the feature, you can merge it into master

Getting Started

- Fork the repo on GitHub
- git clone git@github.com:your_name_here/workbench.git

New Feature or Bug

```
$ git checkout -b my-awesome
$ git push -u origin my-awesome
$ <code for a bit>; git push
$ <code for a bit>; git push
$ tox (this will run all the tests)
```

- Go to github and hit 'New pull request'
- Someone reviews it and says 'AOK'
- Merge the pull request (green button)

Tips

- Any questions/issue please join us on either the Email Forums or Gitter :)

PyPI Checklist (Senior Dev Stuff)

- Spin up a fresh Python Virtual Environment
- Make a git branch called 'v0.2.2-alpha' or whatever

Warning: Make sure workbench/data/memory_images/exemplar4.vmem isn't there, remove if necessary!

```
$ pip install -e .
$ python setup.py sdist
$ cd dist
$ tar xzvf workbench-0.x.y.tar.gz
$ cd workbench-0.x.y/
$ python setup.py install
$ workbench_server
```

- look at output, make sure EVERYTHING comes up okay
- quit workbench_server (ctrl-c in the server window)

```
$ pip install tox
$ tox (pass all tests)
```

- change version in workbench/__init__.py
- change version in setup.py
- Update HISTORY.rst

```
$ python setup.py publish
```

- Spin up another fresh Python Virtual Environment

```
$ pip install workbench --pre
$ workbench_server (in one terminal)
$ pip install pytest-cov
$ cd workbench/workers
$ ./runtests (in another terminal)
```

- Push the version branch

- Go to git do a PR
- Wait for green build and merge
- Create a new release with the same version (v0.2.2-alpha or whatever)
- Claim success!

1.1.6 Credits

Development Lead

- Brian Wylie <https://github.com/brifordwylie>

Contributors

- Ankush Chadda <https://github.com/iamkhush>
- KevtheHermit <https://github.com/kevthehermit>
- Anthony Kasza <https://github.com/anthonykasza>
- Jeffery Baumes <https://github.com/jeffbaumes>
- Ben Mixon-Baca <https://github.com/beenjaminmb>

1.1.7 History

0.1 (2014-06-10)

- Release of workbench for alpha developers and users.

0.1.5 (2014-06-10)

- Release of workbench for alpha developers and users.

0.2.5 (2014-07-07)

- Release of workbench for alpha developers and users.

0.2.6 (2014-07-11)

- Release of workbench for alpha developers and users.

1.1.8 workbench.clients package

Submodules

workbench.clients.customer_report module

This client generates customer reports on all the samples in workbench.

workbench.clients.customer_report.**run**()
This client generates customer reports on all the samples in workbench.

workbench.clients.customer_report.**test**()
Executes test for customer_report.

workbench.clients.help_client module

This client calls a bunch of help commands from workbench

workbench.clients.help_client.**run**()
This client calls a bunch of help commands from workbench

workbench.clients.help_client.**test**()
help_client test

workbench.clients.log_meta_stream module

This client gets metadata about log files.

workbench.clients.log_meta_stream.**run**()
This client gets metadata about log files.

workbench.clients.log_meta_stream.**test**()
Executes log_meta_stream test.

workbench.clients.pcap_bro_indexer module

This client pushes PCAPs -> Bro -> ELS Indexer.

workbench.clients.pcap_bro_indexer.**run**()
This client pushes PCAPs -> Bro -> ELS Indexer.

workbench.clients.pcap_bro_indexer.**test**()
Executes pcap_bro_indexer test.

workbench.clients.pcap_bro_raw module

This client gets the raw bro logs from PCAP files.

workbench.clients.pcap_bro_raw.**run**()
This client gets the raw bro logs from PCAP files.

workbench.clients.pcap_bro_raw.**test**()
Executes pcap_bro_raw test.

workbench.clients.pcap_bro_urls module

This client gets extracts URLs from PCAP files (via Bro logs).

workbench.clients.pcap_bro_urls.**run**()
This client gets extracts URLs from PCAP files (via Bro logs).

workbench.clients.pcap_bro_urls.**test**()
Exexutes pcap_bro_urls test.

workbench.clients.pcap_bro_view module

This client pulls PCAP 'views' (view summarize what's in a sample).

workbench.clients.pcap_bro_view.**run**()
This client pulls PCAP 'views' (view summarize what's in a sample).

workbench.clients.pcap_bro_view.**test**()
pcap_bro_view test

workbench.clients.pcap_meta module

This client pulls PCAP meta data.

workbench.clients.pcap_meta.**run**()
This client pulls PCAP meta data.

workbench.clients.pcap_meta.**test**()
Executes pcap_meta test.

workbench.clients.pcap_meta_indexer module

This client pushes PCAPs -> MetaDaa -> ELS Indexer.

workbench.clients.pcap_meta_indexer.**run**()
This client pushes PCAPs -> MetaDaa -> ELS Indexer.

workbench.clients.pcap_meta_indexer.**test**()
Executes pcap_meta_indexer test.

workbench.clients.pe_indexer module

This client pushes PE Files -> ELS Indexer.

workbench.clients.pe_indexer.**run**()
This client pushes PE Files -> ELS Indexer.

workbench.clients.pe_indexer.**test**()
Executes pe_strings_indexer test.

workbench.clients.pe_peid module

This client looks for PEid signatures in PE Files.

workbench.clients.pe_peid.**run**()
This client looks for PEid signatures in PE Files.

workbench.clients.pe_peid.**test**()
Executes pe_peid test.

workbench.clients.pe_sim_graph module

This client generates a similarity graph from features in PE Files.

workbench.clients.pe_sim_graph.**add_it**(*workbench*, *file_list*, *labels*)
Add the given file_list to workbench as samples, also add them as nodes.

Parameters

- **workbench** – Instance of Workbench Client.
- **file_list** – list of files.
- **labels** – labels for the nodes.

Returns A list of md5s.

workbench.clients.pe_sim_graph.**jaccard_sims**(*feature_list*)
Compute Jaccard similarities between all the observations in the feature list.

Parameters **feature_list** – a list of dictionaries, each having structure as { 'md5' : String, 'features': list of Strings }

Returns list of dictionaries with structure as {'source': md5 String, 'target': md5 String, 'sim': Jaccard similarity Number}

workbench.clients.pe_sim_graph.**jaccard_sim**(*features1*, *features2*)
Compute similarity between two sets using Jaccard similarity.

Parameters

- **features1** – list of PE Symbols.
- **features2** – list of PE Symbols.

Returns Returns an int.

workbench.clients.pe_sim_graph.**run**()
This client generates a similarity graph from features in PE Files.

workbench.clients.upload_dir module

This client pushes a big directory of different files into Workbench.

workbench.clients.upload_dir.**run**()
This client pushes a big directory of different files into Workbench.

workbench.clients.upload_dir.**test**()
Executes file_upload test.

workbench.clients.upload_file module

This client pushes a file into Workbench.

workbench.clients.upload_file.**run**()
This client pushes a file into Workbench.

workbench.clients.upload_file.**test**()
Executes file_upload test.

workbench.clients.workbench_client module

This encapsulates some boilerplate workbench client code.

workbench.clients.workbench_client.**grab_server_args**()
Grab server info from configuration file

workbench.clients.zip_file_extraction module

This client shows workbench extacting files from a zip file.

workbench.clients.zip_file_extraction.**run**()
This client shows workbench extacting files from a zip file.

workbench.clients.zip_file_extraction.**test**()
Executes simple_workbench_client test.

Module contents

Workbench Clients.

1.1.9 workbench.server package

Subpackages

workbench.server.bro package

Submodules

workbench.server.bro.bro_log_reader module This module handles the mechanics around easily pulling in Bro Log data.

The read_log method is a generator (in the python sense) for rows in a Bro log, because of this, it's memory efficient and does not read the entire file into memory.

class workbench.server.bro.bro_log_reader.**BroLogReader**(*convert_datetimes=True*)
Bases: object

This class implements a python based Bro Log Reader.

Init for BroLogReader.

read_log(*logfile*)
The read_log method returns a memory efficient generator for rows in a Bro log.

Usage: rows = my_bro_reader.read_log(logfile) for row in rows:

do something with row

Parameters **logfile** – The Bro Log file.

Module contents

Submodules

workbench.server.data_store module

DataStore class for WorkBench.

class workbench.server.data_store.**DataStore**(*uri='mongodb://localhost/workbench'*, *database='workbench'*, *worker_cap=0*, *samples_cap=0*)

Bases: object

DataStore for Workbench.

Currently tied to MongoDB but making this class 'abstract' should be straightforward and we could think about using another backend.

Initialization for the Workbench data store class.

Parameters

- **uri** – Connection String for DataStore backend.
- **database** – Name of database.
- **worker_cap** – MBs in the capped collection.
- **samples_cap** – MBs of sample to be stored.

get_uri()
Return the uri of the data store.

store_sample(*filename*, *sample_bytes*, *type_tag*)
Store a sample into the datastore.

Parameters

- **filename** – Name of the file.
- **sample_bytes** – Actual bytes of sample.
- **type_tag** – Type of sample ('exe','pcap','pdf','json','swf', or ...).

Returns Digest md5 digest of the sample.

sample_storage_size()
Get the storage size of the samples storage collection.

expire_data()
Expire data within the samples collection.

clean_for_serialization(*data*)
Clean data in preparation for serialization.

Deletes items having key either a BSON, datetime, dict or a list instance, or starting with __.

Parameters **data** – Sample data to be serialized.

Returns Cleaned data dictionary.

clean_for_storage(*data*)
Clean data in preparation for storage.

Deletes items with key having a '.' or is '_id'. Also deletes those items whose value is a dictionary or a list.

Parameters **data** – Sample data dictionary to be cleaned.

Returns Cleaned data dictionary.

get_sample(*md5*)
Get the sample from the data store.

This method first fetches the data from datastore, then cleans it for serialization and then updates it with 'raw_bytes' item.

Parameters **md5** – The md5 digest of the sample to be fetched from datastore.

Returns The sample dictionary.

Raises `RuntimeError` – Either Sample is not found or the gridfs file is missing.

get_sample_window(*type_tag*, *size=10*)
Get a window of samples not to exceed size (in MB).

Parameters

- **type_tag** – Type of sample ('exe','pcap','pdf','json','swf', or ...).
- **size** – Size of samples in MBs.

Returns a list of md5s.

has_sample(*md5*)
Checks if data store has this sample.

Parameters **md5** – The md5 digest of the required sample.

Returns True if sample with this md5 is present, else False.

list_samples(*predicate={}*)
List all samples that meet the predicate or all if predicate is not specified.

Parameters **predicate** – Match samples against this predicate (or all if not specified)

Returns List of dictionaries with matching samples {'md5':md5, 'filename': 'foo.exe', 'type_tag': 'exe'}

store_work_results(*results*, *collection*, *md5*)
Store the output results of the worker.

Parameters

- **results** – a dictionary.
- **collection** – the database collection to store the results in.
- **md5** – the md5 of sample data to be updated.

get_work_results(*collection*, *md5*)
Get the results of the worker.

Parameters

- **collection** – the database collection storing the results.
- **md5** – the md5 digest of the data.

Returns Dictionary of the worker result.

all_sample_md5s(*type_tag=None*)
Return a list of all md5 matching the type_tag ('exe','pdf', etc).

Parameters **type_tag** – the type of sample.

Returns a list of matching samples.

clear_db()
Drops the entire workbench database.

periodic_ops()
Run periodic operations on the the data store.

Operations like making sure collections are capped and indexes are set up.

to_unicode(*s*)
Convert an elementary datatype to unicode.

Parameters **s** – the datatype to be unicoded.

Returns Unicoded data.

data_to_unicode(*data*)
Recursively convert a list or dictionary to unicode.

Parameters **data** – The data to be unicoded.

Returns Unicoded data.

workbench.server.els_indexer module

ELSIndexer class for WorkBench.

class workbench.server.els_indexer.**ELSStubIndexer**(*hosts='[{"host": "localhost", "port": 9200}]'*)
Bases: object

ELS Stub.

Stub Indexer Initialization.

index_data(*data*, *index_name*, *doc_type*)
Index data in Stub Indexer.

search(*index_name*, *query*)
Search in Stub Indexer.

class workbench.server.els_indexer.**ELSIndexer**(*hosts=None*)
Bases: object

ELSIndexer class for WorkBench.

Initialization for the Elastic Search Indexer.

Parameters **hosts** – List of connection settings.

index_data(*data*, *index_name*, *doc_type*)
Take an arbitrary dictionary of data and index it with ELS.

Parameters

- **data** – data to be Indexed. Should be a dictionary.
- **index_name** – Name of the index.
- **doc_type** – The type of the document.

Raises RuntimeError – When the Indexing fails.

search(*index_name*, *query*)
Search the given index_name with the given ELS query.

Parameters

- **index_name** – Name of the Index
- **query** – The string to be searched.

Returns List of results.

Raises RuntimeError – When the search query fails.

workbench.server.neo_db module

NeoDB class for WorkBench.

class workbench.server.neo_db.**NeoDBStub**(*uri='http://localhost:7474/db/data'*)
Bases: object

NeoDB Stub.

NeoDB Stub.

add_node(*node_id*, *name*, *labels*)
NeoDB Stub.

has_node(*node_id*)
NeoDB Stub.

add_rel(*source_node_id*, *target_node_id*, *rel*)
NeoDB Stub.

clear_db()
NeoDB Stub.

class workbench.server.neo_db.**NeoDB**(*uri='http://localhost:7474/db/data'*)
Bases: object

NeoDB indexer for Workbench.

Initialization for NeoDB indexer.

Parameters **uri** – The uri to connect NeoDB.

Raises RuntimeError – When connection to NeoDB failed.

add_node(*node_id*, *name*, *labels*)
Add the node with name and labels.

Parameters

- **node_id** – Id for the node.
- **name** – Name for the node.
- **labels** – Label for the node.

Raises NotImplementedError – When adding labels is not supported.

has_node(*node_id*)
Checks if the node is present.

Parameters **node_id** – Id for the node.

Returns True if node with node_id is present, else False.

add_rel(*source_node_id*, *target_node_id*, *rel*)
Add a relationship between nodes.

Parameters

- **source_node_id** – Node Id for the source node.
- **target_node_id** – Node Id for the target node.
- **rel** – Name of the relationship 'contains'

clear_db()
Clear the Graph Database of all nodes and edges.

workbench.server.plugin_manager module

A simple plugin manager. Rolling my own for three reasons: 1) Environmental scan did not give me quite what I wanted. 2) The super simple examples didn't support automatic/dynamic loading. 3) I kinda wanted to understand the process :)

`workbench.server.plugin_manager.test()`
Executes plugin_manager.py test.

workbench.server.workbench module

Workbench: Open Source Security Framework

class `workbench.server.workbench.WorkBench`(*store_args=None*, *els_hosts=None*, *neo_uri=None*)

Bases: `object`

Workbench: Open Source Security Framework.

Initialize the Framework.

Parameters

- **store_args** – Dictionary with keys uri,database,samples_cap, worker_cap.
- **els_hosts** – The address where Elastic Search Indexer is running.
- **neo_uri** – The address where Neo4j is running.

store_sample(*filename*, *input_bytes*, *type_tag*)
Store a sample into the DataStore. :param filename: name of the file (used purely as meta data not for lookup) :param input_bytes: the actual bytes of the sample e.g. f.read() :param type_tag: ('exe','pcap','pdf','json','swf', or ...)

Returns the md5 of the sample.

get_sample(*md5*)
Get a sample from the DataStore. :param md5: the md5 of the sample

Returns A dictionary of meta data about the sample which includes a ['raw_bytes'] key that contains the raw bytes.

get_sample_window(*type_tag*, *size*)
Get a sample from the DataStore. :param type_tag: the type of samples ('pcap','exe','pdf') :param size: the size of the window in MegaBytes (10 = 10MB)

Returns A list of md5s representing the newest samples within the size window

has_sample(*md5*)
Do we have this sample in the DataStore. :param md5: the md5 of the sample

Returns True or False

list_samples(*predicate={}*)
List all samples that meet the predicate or all if predicate is not specified.

Parameters **predicate** – Match samples against this predicate (or all if not specified)

Returns List of dictionaries with matching samples {'md5':md5, 'filename': 'foo.exe', 'type_tag': 'exe'}

stream_sample = <Mock name='mock.stream()' id='139991185865680'>

guess_type_tag(*input_bytes*)
Try to guess the type_tag for this sample

index_sample(*md5*, *index_name*)
Index a stored sample with the Indexer. :param md5: the md5 of the sample :param index_name: the name of the index

Returns Nothing

index_worker_output(*worker_name*, *md5*, *index_name*, *subfield*)
Index worker output with the Indexer. :param worker_name: 'strings', 'pe_features', whatever :param md5: the md5 of the sample :param index_name: the name of the index :param subfield: index just this subfield (None for all)

Returns Nothing

search(*index_name*, *query*)
Search a particular index in the Indexer :param index_name: the name of the index :param query: the query against the index

Returns All matches to the query

add_node(*node_id*, *name*, *labels*)
Add a node to the graph with name and labels. :param node_id: the unique node_id e.g. 'www.evil4u.com' :param name: the display name of the node e.g. 'evil4u' :param labels: a list of labels e.g. ['domain','evil']

Returns Nothing

has_node(*node_id*)
Does the Graph DB have this node :param node_id: the unique node_id e.g. 'www.evil4u.com'

Returns True/False

add_rel(*source_id*, *target_id*, *rel*)
Add a relationship: source, target must already exist (see add_node) 'rel' is the name of the relationship 'contains' or whatever. :param source_id: the unique node_id of the source :param target_id: the unique node_id of the target :param rel: name of the relationship

Returns Nothing

clear_graph_db()
Clear the Graph Database of all nodes and edges.

Returns Nothing

clear_db()
Clear the Main Database of all samples and worker output.

Returns Nothing

work_request(*worker_name*, *md5*, *subkeys=None*)
Make a work request for an existing stored sample. :param worker_name: 'strings', 'pe_features', whatever :param md5: the md5 of the sample :param subkeys: just return a subfield e.g. 'foo' or 'foo.bar' (None for all)

Returns The output of the worker or just the subfield of the worker output

batch_work_request = <Mock name='mock.stream()' id='139991185865680'>

store_sample_set(*md5_list*)
Store a sample set (which is just a list of md5s).

Note: All md5s must already be in the data store.

Parameters **md5_list** – a list of the md5s in this set (all must exist in data store)

Returns The md5 of the set (the actual md5 of the set

get_sample_set(*md5*)
Store a sample set (which is just a list of md5s).

Parameters **md5_list** – a list of the md5s in this set (all must exist in data store)

Returns The md5 of the set (the actual md5 of the set

stream_sample_set = <Mock name='mock.stream()' id='139991185865680'>

get_datastore_uri()
Gives you the current datastore URL.

Returns The URI of the data store currently being used by Workbench

help(*cli=False*)
Returns help commands

help_basic(*cli=False*)
Returns basic help commands

help_commands(*cli=False*)
Returns a big string of Workbench commands and signatures

help_command(*command*, *cli=False*)
Returns a specific Workbench command and docstring

help_workers(*cli=False*)
Returns a big string of the loaded Workbench workers and their dependencies

help_worker(*worker*, *cli=False*)
Returns a specific Workbench worker and docstring

help_advanced(*cli=False*)
Returns advanced help commands

help_everything(*cli=False*)
Returns advanced help commands

list_all_commands()
Returns a list of all the Workbench commands

list_all_workers()
List all the currently loaded workers

worker_info(*worker_name*)
Get the information about this worker

test_worker(*worker_name*)
Run the test for a specific worker

workbench.server.workbench.**run**()
Run the workbench server

workbench.server.workbench.**test**()

Module contents

1.1.10 workbench.workers package

Subpackages

workbench.workers.rekall_adapter package

Submodules

workbench.workers.rekall_adapter.rekall_adapter module rekall_adapter: Helps Workbench utilize the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :).

workbench.workers.rekall_adapter.rekall_adapter.**gsleep**()

class workbench.workers.rekall_adapter.rekall_adapter.**RekallAdapter**(*raw_bytes*)
Bases: object

RekallAdapter: Helps utilize the Rekall Memory Forensic Framework.

Initialization.

get_session()

get_renderer()

class workbench.workers.rekall_adapter.rekall_adapter.**MemSession**(*raw_bytes*)
Bases: object

MemSession: Helps utilize the Rekall Memory Forensic Framework.

Create a Rekall session from raw_bytes.

get_session()
Get the current session handle.

workbench.workers.rekall_adapter.rekall_adapter.**test**()
rekall_adapter.py: Test.

Module contents

Submodules

workbench.workers.json_meta module

JSON Meta worker

class workbench.workers.json_meta.**JSONMetaData**
Bases: object

This worker computes meta-data for json files.

Initialization

dependencies = ['sample', 'meta']

execute(*input_data*)

workbench.workers.json_meta.**test**()
json_meta.py: Test

workbench.workers.log_meta module

Logfile Meta worker

class workbench.workers.log_meta.**LogMetaData**
Bases: object

This worker computes a meta-data for log files.

Initialization

dependencies = ['sample', 'meta']

execute(*input_data*)

workbench.workers.log_meta.**test**()
log_meta.py: Unit test

workbench.workers.mem_base module

Memory Image base worker. This worker utilizes the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :)

class workbench.workers.mem_base.**MemoryImageBase**
Bases: object

This worker computes meta-data for memory image files.

Initialization

dependencies = ['sample']

set_plugin_name(*name*)
Set the name of the plugin to be used

execute(*input_data*)
Execute method

workbench.workers.mem_base.**test**()
mem_base.py: Test

workbench.workers.mem_connscan module

Memory Image ConnScan worker. This worker utilizes the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :)

class workbench.workers.mem_connscan.**MemoryImageConnScan**
Bases: workbench.workers.mem_base.MemoryImageBase

This worker computes connscan-data for memory image files.

Initialization

dependencies = ['sample']

execute(*input_data*)

workbench.workers.mem_dlllist module

Memory Image DllList worker. This worker utilizes the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :)

class workbench.workers.mem_dlllist.**MemoryImageDllList**
Bases: workbench.workers.mem_base.MemoryImageBase

This worker computes dlllist for memory image files.

Initialization

dependencies = ['sample']

static safe_key(*key*)

execute(*input_data*)

workbench.workers.mem_meta module

Memory Image Meta worker. This worker utilizes the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :)

class workbench.workers.mem_meta.**MemoryImageMeta**
Bases: workbench.workers.mem_base.MemoryImageBase

This worker computes meta-data for memory image files.

Initialization

dependencies = ['sample']

execute(*input_data*)

workbench.workers.mem_procdump module

Memory Image ProcDump worker. This worker utilizes the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :)

class workbench.workers.mem_procdump.**MemoryImageProcDump**
Bases: object

This worker dumps process pe files from memory image files.

Initialization

dependencies = ['sample']

execute(*input_data*)
Execute method

make_temp_directory(**args*, ***kwds*)

__del__()
Class Cleanup

workbench.workers.mem_pslist module

Memory Image PSList worker. This worker utilizes the Rekall Memory Forensic Framework. See Google Github: http://github.com/google/rekall All credit for good stuff goes to them, all credit for bad stuff goes to us. :)

class workbench.workers.mem_pslist.**MemoryImagePSList**
Bases: workbench.workers.mem_base.MemoryImageBase

This worker computes pslist-data for memory image files.

Initialization

dependencies = ['sample']

execute(*input_data*)

workbench.workers.meta module

Meta worker

class workbench.workers.meta.**MetaData**
Bases: object

This worker computes meta data for any file type.

Initialization

dependencies = ['sample']

execute(*input_data*)
This worker computes meta data for any file type.

workbench.workers.meta.**test**()
meta.py: Unit test

workbench.workers.meta_deep module

MetaDeep worker

class workbench.workers.meta_deep.**MetaDeepData**
Bases: object

This worker computes deeper meta-data

Initialization

dependencies = ['sample', 'meta']

execute(*input_data*)

workbench.workers.meta_deep.**test**()
meta_deep.py: Unit test

workbench.workers.pcap_bro module

PcapBro worker

class workbench.workers.pcap_bro.**PcapBro**
Bases: object

This worker runs Bro scripts on a pcap file

dependencies = ['sample']

get_bro_script_path()

setup_pcap_inputs(*input_data*)
Write the PCAPs to disk for Bro to process and return the pcap filenames

execute(*input_data*)
Execute

subprocess_manager(*exec_args*)
Bro subprocess manager

make_temp_directory(**args*, ***kwds*)
Bro temporary directory context manager

__del__()
Class Cleanup

workbench.workers.pcap_bro.**test**()
pcap_bro.py: Unit test

workbench.workers.pcap_graph module

pcap_graph worker

class workbench.workers.pcap_graph.**PcapGraph**
Bases: object

This worker generates a graph from a PCAP (depends on Bro)

Initialization

dependencies = ['pcap_bro']

add_node(*node_id*, *name*, *labels*)
Cache aware add_node

add_rel(*source_id*, *target_id*, *rel*)
Cache aware add_rel

execute(*input_data*)
Okay this worker is going build graphs from PCAP Bro output logs

conn_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro conn.log)

http_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro http.log)

dns_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro dns.log)

weird_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro weird.log)

files_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro dns.log)

__del__()
Class Cleanup

workbench.workers.pcap_graph.**test**()
pcap_graph.py: Unit test

workbench.workers.pcap_http_graph module

pcap_http_graph worker

class workbench.workers.pcap_http_graph.**PcapHTTPGraph**
Bases: object

This worker generates a graph from a PCAP (depends on Bro)

Initialization

dependencies = ['pcap_bro']

add_node(*node_id*, *name*, *labels*)
Cache aware add_node

add_rel(*source_id*, *target_id*, *rel*)
Cache aware add_rel

execute(*input_data*)
Okay this worker is going build graphs from PCAP Bro output logs

http_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro http.log)

weird_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro weird.log)

files_log_graph(*stream*)
Build up a graph (nodes and edges from a Bro dns.log)

__del__()
Class Cleanup

workbench.workers.pcap_http_graph.**test**()
pcap_http_graph.py: Unit test

workbench.workers.pe_classifier module

PE Classify worker (just a placeholder, not a real classifier at this point)

class workbench.workers.pe_classifier.**PEFileClassify**
Bases: object

This worker classifies PEFiles as Evil or Benign (TOY not a real classifier at this point)

Initialization

dependencies = ['pe_features', 'pe_indicators']

execute(*input_data*)
This worker classifies PEFiles as Evil or Benign (TOY not a real classifier at this point)

workbench.workers.pe_classifier.**test**()
pe_classifier.py: Unit test

workbench.workers.pe_deep_sim module

PE SSDeep Similarity worker

class workbench.workers.pe_deep_sim.**PEDeepSim**
Bases: object

This worker computes fuzzy matches between samples with ssdeep

dependencies = ['meta_deep']

execute(*input_data*)
Execute method

__del__()
Class Cleanup

workbench.workers.pe_deep_sim.**test**()
pe_deep_sim.py: Unit test

workbench.workers.pe_features module

PE Features worker. This class pulls static features out of a PE file using the python pefile module.

class workbench.workers.pe_features.**PEFileWorker**(*verbose=False*)
Bases: object

Create instance of PEFileWorker class. This class pulls static features out of a PE file using the python pefile module.

Init method

dependencies = ['sample']

execute(*input_data*)
Process the input bytes with pefile

set_dense_features(*dense_feature_list*)
Set the dense feature list that the Python pefile module should extract. This is really just sanity check functionality, meaning that these are the features you are expecting to get, and a warning will spit out if you don't get some of these.

get_dense_features()
Set the dense feature list that the Python pefile module should extract.

set_sparse_features(*sparse_feature_list*)
Set the sparse feature list that the Python pefile module should extract. This is really just sanity check functionality, meaning that these are the features you are expecting to get, and a warning will spit out if you don't get some of these.

get_sparse_features()
Set the sparse feature list that the Python pefile module should extract.

static open_using_pefile(*input_name*, *input_bytes*)
Open the PE File using the Python pefile module.

extract_features_using_pefile(*pef*)
Process the PE File using the Python pefile module.

workbench.workers.pe_features.**convert_to_utf8**(*string*)
Convert string to UTF8

workbench.workers.pe_features.**convert_to_ascii_null_term**(*string*)
Convert string to Null terminated ascii

workbench.workers.pe_features.**test**()
pe_features.py: Test

workbench.workers.pe_indicators module

This python class codifies a bunch of rules around suspicious static features in a PE File. The rules don't indicate malicious behavior they simply flag things that may be used by a malicious binary. Many of the indicators used were inspired by the material in the 'Practical Malware Analysis' book by Sikorski and Honig, ISBN-13: 978-1593272906 (available on Amazon :)

Description:

> PE_WARNINGS = PE module warnings verbatim MALFORMED = the PE file is malformed COMMUNICATION = network activities CREDENTIALS = activities associated with elevating or attaining new privileges KEYLOGGING = activities associated with keylogging SYSTEM_STATE = file system or registry activities SYSTEM_PROBE = getting information from the local system (file system, OS config) SYSTEM_INTEGRITY = compromises the security state of the local system PROCESS_MANIPULATION = indicators associated with process manipulation/injection PROCESS_SPAWN = indicators associated with creating a new process STEALTH_LOAD = indicators associated with loading libraries, resources, etc in a sneaky way ENCRYPTION = any indicators related to encryption COM_SERVICES = COM functionality or running as a service ANTI_DEBUG = anti-debugging indicators

class workbench.workers.pe_indicators.**PEIndicators**
Bases: object

Create instance of Indicators class. This class uses the static features from the pefile module to look for weird stuff.

Note: All methods that start with 'check' will be automatically included as part of the checks that happen when 'execute' is called.

Init method of the Indicators class.

dependencies = ['sample']

execute(*input_data*)
Execute the PEIndicators worker

check_corrupted_imports()
Various ways the imports table might be corrupted.

check_checksum_is_zero()
Checking for a checksum of zero

check_checksum_mismatch()
Checking for a checksum that doesn't match the generated checksum

check_empty_section_name()
Checking for an empty section name

check_nonstandard_section_name()
Checking for an non-standard section name

check_image_size_incorrect()
Checking if the reported image size matches the actual image size

check_overlapping_headers()
Checking if pefile module reported overlapping header

check_section_unaligned()
Checking if any of the sections are unaligned

check_section_oversized()
Checking if any of the sections go past the total size of the image

check_dll_with_no_exports()
Checking if the PE is a DLL with no exports

check_communication_imports()
Checking if the PE imports known communication methods

check_elevating_privs_imports()
Checking if the PE imports known methods associated with elevating or attaining new privileges

check_keylogging_imports()
Checking if the PE imports known methods associated with elevating or attaining new privileges

check_system_state_imports()
Checking if the PE imports known methods associated with changing system state

check_system_probe_imports()
Checking if the PE imports known methods associated with probing the system

check_system_integrity_imports()
Checking if the PE imports known methods associated with system security or integrity

check_crypto_imports()
Checking if the PE imports known methods associated with encryption

check_anti_debug_imports()
Checking if the PE imports known methods associated with anti-debug

check_com_service_imports()
Checking if the PE imports known methods associated with COM or services

check_process_manipulation()
Checking if the PE imports known methods associated with process manipulation/injection

check_process_spawn()
Checking if the PE imports known methods associated with spawning a new process

check_stealth_load()
Checking if the PE imports known methods associated with loading libraries, resources, etc in a sneaky way

check_invalid_entry_point()
Checking the PE File warning for an invalide entry point

check_exports()
This is just a stub function right now, might be useful later

workbench.workers.pe_indicators.**convert_to_ascii_null_term**(*string*)
Convert string to null terminated ascii string

workbench.workers.pe_indicators.**test**()
pe_indicators.py: Unit test

workbench.workers.pe_peid module

PE peid worker, uses the peid_userdb.txt database of signatures

class workbench.workers.pe_peid.**PEIDWorker**
Bases: object

This worker looks up pe_id signatures for a PE file.

dependencies = ['sample']

execute(*input_data*)
Execute the PEIDWorker

peid_features(*pefile_handle*)
Get features from PEid signature database

workbench.workers.pe_peid.**test**()
pe_peid.py: Unit test

workbench.workers.strings module

Strings worker

class workbench.workers.strings.**Strings**
Bases: object

This worker extracts all the strings from any type of file

Initialize the Strings worker

dependencies = ['sample']

execute(*input_data*)
Execute the Strings worker

workbench.workers.strings.**test**()
strings.py: Unit test

workbench.workers.swf_meta module

SWFMeta worker: This is a stub the real class (under the experimental directory has too many dependencies)

class workbench.workers.swf_meta.**SWFMeta**
Bases: object

This worker computes a bunch of meta-data about a SWF file

dependencies = ['sample', 'meta']

execute(*input_data*)
Execute the SWFMeta worker

workbench.workers.swf_meta.**test**()
swf_meta.py: Unit test

workbench.workers.unzip module

Unzip worker

class workbench.workers.unzip.**Unzip**
Bases: object

This worker unzips a zipped file

dependencies = ['sample']

execute(*input_data*)
Execute the Unzip worker

__del__()
Class Cleanup

workbench.workers.unzip.**test**()
unzip.py: Unit test

workbench.workers.url module

URLS worker: Tries to extract URL from strings output

class workbench.workers.url.**URLS**
Bases: object

This worker looks for url patterns in strings output

Initialize the URL worker

dependencies = ['strings']

execute(*input_data*)
Execute the URL worker

workbench.workers.url.**test**()
url.py: Unit test

workbench.workers.view module

view worker

class workbench.workers.view.**View**
Bases: object

View: Generates a view for any file type

dependencies = ['meta']

execute(*input_data*)

__del__()
Class Cleanup

workbench.workers.view.**test**()
view.py: Unit test

workbench.workers.view_customer module

view_customer worker

class workbench.workers.view_customer.**ViewCustomer**
Bases: object

ViewCustomer: Generates a customer usage view.

dependencies = ['meta']

execute(*input_data*)
Execute Method

workbench.workers.view_customer.**test**()
view_customer.py: Unit test

workbench.workers.view_log_meta module

view_log_meta worker

class workbench.workers.view_log_meta.**ViewLogMeta**
Bases: object

ViewLogMeta: Generates a view for meta data on the sample

dependencies = ['log_meta']

execute(*input_data*)
Execute the ViewLogMeta worker

workbench.workers.view_log_meta.**test**()
view_log_meta.py: Unit test

workbench.workers.view_memory module

view_memory worker

class workbench.workers.view_memory.**ViewMemory**
Bases: object

ViewMemory: Generates a view for meta data on the sample

dependencies = ['mem_connscan', 'mem_meta', 'mem_procdump', 'mem_pslist']

execute(*input_data*)
Execute the ViewMemory worker

workbench.workers.view_memory.**test**()
view_memory.py: Unit test

workbench.workers.view_meta module

view_meta worker

class workbench.workers.view_meta.**ViewMetaData**
Bases: object

ViewMetaData: Generates a view for meta data on the sample

dependencies = ['meta']

execute(*input_data*)
Execute the ViewMetaData worker

workbench.workers.view_meta.**test**()
view_meta.py: Unit test

workbench.workers.view_pcap module

view_pcap worker

class workbench.workers.view_pcap.**ViewPcap**
Bases: object

ViewPcap: Generates a view for a pcap sample (depends on Bro)

dependencies = ['pcap_bro']

execute(*input_data*)
Execute

__del__()
Class Cleanup

workbench.workers.view_pcap.**test**()
view_pcap.py: Unit test

workbench.workers.view_pcap_details module

view_pcap_details worker

class workbench.workers.view_pcap_details.**ViewPcapDetails**
Bases: object

ViewPcapDetails: Generates a view for a pcap sample (depends on Bro)

Initialization of ViewPcapDetails

dependencies = ['view_pcap']

execute(*input_data*)
ViewPcapDetails execute method

__del__()
Class Cleanup

workbench.workers.view_pcap_details.**test**()
view_pcap_details.py: Unit test

workbench.workers.view_pdf module

view_pdffile worker

class workbench.workers.view_pdf.**ViewPDFFile**
Bases: object

ViewPDFFile: Generates a view for PDF files

dependencies = ['meta', 'strings']

execute(*input_data*)
Execute the ViewPDF worker

workbench.workers.view_pdf.**test**()
' view_pdf.py: Unit test

workbench.workers.view_pe module

view_pe worker

class workbench.workers.view_pe.**ViewPEFile**
Bases: object

Generates a high level summary view for PE files that incorporates a large set of workers

dependencies = ['meta', 'strings', 'pe_peid', 'pe_indicators', 'pe_classifier']

execute(*input_data*)
Execute the ViewPEFile worker

static safe_get(*data*, *key_list*)
Safely access dictionary keys when plugin may have failed

workbench.workers.view_pe.**test**()
view_pe.py: Unit test

workbench.workers.view_zip module

view_zip worker

class workbench.workers.view_zip.**ViewZip**
Bases: object

ViewZip: Generates a view for Zip files

dependencies = ['meta', 'unzip']

execute(*input_data*)
Execute the ViewZip worker

__del__()
Class Cleanup

workbench.workers.view_zip.**test**()
– view_zip.py test –

workbench.workers.vt_query module

VTQuery worker

class workbench.workers.vt_query.**VTQuery**
Bases: object

This worker query Virus Total, an apikey needs to be provided

VTQuery Init

dependencies = ['meta']

execute(*input_data*)
Execute the VTQuery worker

workbench.workers.vt_query.**test**()
– vt_query.py test –

workbench.workers.yara_sigs module

Yara worker

class workbench.workers.yara_sigs.**YaraSigs**
Bases: object

This worker check for matches against yara sigs. Output keys: [matches:list of matches]

dependencies = ['sample']

get_yara_rules()
Recursively traverse the yara/rules directory for rules

execute(*input_data*)
yara worker execute method

workbench.workers.yara_sigs.**test**()
yara.py: Unit test

Module contents

W