stoQ Documentation

Release 1.0.2

PUNCH Cyber Analytics Group

Dec 18, 2018

Contents

Quick Links

CHAPTER 1

Installing

1.1 Installation Script

If using Ubuntu, Redhat 7, or CentOS, installation of the core framework and plugins can be installed utilizing the installation script provided with the framework.:

```
git clone https://github.com/PUNCH-Cyber/stoq.git
cd stoq/
bash install.sh
```

Note: **stoQ** has not been tested on other operating systems, however, if the required packages are available it should work without issue.

1.2 Detailed Ubuntu Installation

1.2.1 Core Requirements

Install the core requirements via apt-get and pip:

```
apt-add-repository -y multiverse
sudo apt-get install automake build-essential cython autoconf  \
                     python3 python3-dev python3-setuptools \
                     libyaml-dev libffi-dev libfuzzy-dev \
                     libxml2-dev libxslt1-dev libz-dev p7zip-full \
                     p7zip-rar unace-nonfree libssl-dev libmagic-dev
sudo easy_install3 pip
```

Define an environment variable for where **stoQ** should be setup, *STOQ_HOME*. If none is setup *$HOME/.stoq* will be used. For the purpose of this installation process, we will use *$STOQ_HOME* in all installation commands that require it:

```
export STOQ_HOME=/usr/local/stoq
```

stoQ does not require any special permissions to run. For security reasons, it is recommended that **stoQ** is run as a non-privileged user. To create a **stoQ** user, run:

```
sudo groupadd -r stoq
sudo useradd -r -c stoQ -g stoq -d $STOQ_HOME stoq
chown -R stoq:stoq $STOQ_HOME
```

It is recommended to install **stoQ** within a virtualenv. This is however completely optional. In order to setup the virtualenv, the following should be completed:

```
sudo pip3 install virtualenv
virtualenv $STOQ_HOME/.stoq-pyenv
source $STOQ_HOME/.stoq-pyenv/bin/activate
```

Install the latest version of yara from https://plusvic.github.io/yara/

Once the virtualenv has been activated and yara is installed, we can install the core **stoQ** requirements:

```
python setup.py install
```

Note: stoQ will install yara-python from pip, however, there is at least one issue (https://github.com/VirusTotal/yara-python/issues/28) that may cause your ruleset to fail. It is recommend that yara-python be install manually with: `pip3 install --global-option="build" --global-option="--dynamic-linking" yara-python`

Copy **stoQ** configuration file and the default dispatcher.yar to **stoQ**'s home directory:

```
cp extras/* $STOQ_HOME
```

The core framework for **stoQ** should now be installed. We can use **stoQ**'s plugin installation feature to handle this. First, we will need to clone **stoQ**'s public plugin repository:

```
git clone https://github.com/PUNCH-Cyber/stoq-plugins-public.git /tmp/stoq-plugins-
↪public
```

Plugins can be installed manually using `stoq install /path/to/plugin`, or, we can install all of the publicly available plugins using a simple script:

```
#!/bin/bash
for category in connector decoder extractor carver source reader worker;
do
    for plugin in `ls /tmp/stoq-plugins-public/$category`;
    do
        stoq install /tmp/stoq-plugins-public/$category/$plugin
    done
done
```

Note:

- ***xorsearch* requires XORsearch to be installed** http://blog.didierstevens.com/programs/xorsearch/
- ***exif* requires ExifTool to be installed** http://www.sno.phy.queensu.ca/~phil/exiftool/
- ***tika* requires that Apache Tika be installed** https://tika.apache.org/download.html

- ***clamav* requires that a ClamAV daemon be installed** http://www.clamav.net/

1.2.2 Additional Plugins

There are several other plugins that are available in the *stoQ* public plugin repository at https://github.com/PUNCH-Cyber/stoq-plugins-public

1.3 Supervisord

stoQ can easily be added to supervisord for running as a system service in daemon mode. In our example, let's say that we want to use the yara and exif plugins to monitor RabbitMQ and save any results into MongoDB. We've installed **stoQ** into /usr/local/stoq and our python virtual environment is in `/usr/local/stoq/.stoq-pyenv`. First, let's install the supervisor Ubuntu package:

```
sudo apt-get install supervisor
```

Now, let's create a new file in `/etc/supervisor/conf.d` named `stoq.conf` with the below content:

```
[program:exif]
command=/usr/local/stoq/.stoq-pyenv/bin/stoq %(program_name)s -I rabbitmq -C mongodb
process_name=%(program_name)s_%(process_num)02d
directory=/usr/local/stoq
autostart=true
autorestart=true
startretries=3
numprocs=1
user=stoq

[program:yara]
command=/usr/local/stoq/.stoq-pyenv/bin/stoq  %(program_name)s -I rabbitmq -C mongodb
process_name=%(program_name)s_%(process_num)02d
directory=/usr/local/stoq
autostart=true
autorestart=true
startretries=3
numprocs=1
user=stoq
```

Then, simply restart supervisord:

```
supervisorctl reload
```

Note: If supervisorctl fails, ensure that the supervisor service is running `service supervisor start`

You should now have two **stoQ** workers running, monitoring their RabbitMQ queue, and saving their results into your MongoDB instance.

1.4 Vagrant

If testing **stoQ** is something you are interested in doing, you can use Vagrant to setup a simple instance.

First, install Vagrant from https://www.vagrantup.com/downloads, then, install VirtualBox from https://www.virtualbox.org/wiki/Downloads.

Once the prerequisites are installed, download the Ubuntu box:

```
vagrant box add ubuntu/xenial64
```

Next, create a new directory named `stoq` and save the Vagrantfile in it:

```
wget -O Vagrantfile https://raw.githubusercontent.com/PUNCH-Cyber/stoq/master/
↪Vagrantfile
```

Now, let's bring up the Vagrant box:

```
vagrant up
```

Log into the new box:

```
vagrant ssh
```

Switch to the `stoq` user:

```
sudo su - stoq
```

Your newly installed **stoQ** instance is now available in `/usr/local/stoq`.

All done!

CHAPTER 2

Plugin Development

2.1 Overview

There are several plugin categories that are available currently:

- worker
- connector
- reader
- source
- extractor
- carver
- decoder
- decorator

The **worker** plugin category is for the plugins that will produce data from payloads and provide the results back to the framework for output. Once the **worker** plugin is complete, the framework will want to handle the results in some fashion. This is where the **connector** plugins come into play. Once the results have been provided back to the framework, the connector is then called. The **connector** plugins may also be used for file archiving if supported within the plugin. One may save a file using a connector by simply calling `save()` with the `archive=True` option. Conversely, in order to retrieve a file from MongoDB's GridFS, one simply would call `get_file()`. **Reader** plugins are used to enrich data for worker plugins such as indicator extraction, STIX support, or a multitude of other enhancements. **Source** plugins handle the messaging and queueing of objects that the worker should handle. For instance, monitoring a directory for new files or AMQP. **Extractor** plugins handle various tasks such as decompressing zip files and deflating pdf streams. **Carver** plugins are used to carve content out of payloads (e.g., SWF streams out of DOC files, PE out of RTF, etc...). **Decoder** plugins provide the capability to automatically decode a payload, such as XOR, ROR, and base64. **Decorator** plugins allow for post processing of results from **stoQ** before being saved or returned.

2.2 Configuration

Each plugin has it's own configuration file ending in *.stoq*. Upon initialization of the plugin, the configuration options within the file will be loaded and made available to the worker object.

At a minimum, the below configuration options are required for all plugins.

```
[Core]
# Name of plugin. stoQ will use this when calling the plugin.
Name = basicplugin
# Name of the .py file for this plugin
Module = basicplugin

[Documentation]
Author = Joe Stoq
Version = 0.1
Website = https://github.com/PUNCH-Cyber/stoq
Description = Basic Plugin Example
```

If a plugin requires additional configuration parameters, they can be added to the `[options]` section and will be made available via the plugin object. For example, if we have defined our plugin object as `plugin`, we can access the `hashpayload` attribute by calling `self.hashpayload`.

```
[options]
hashpayload = True
saveresults = True
max_tlp = red
max_stoq_version = 0.10.3
min_stoq_version = 0.9
ratelimit = 1/5
```

Note: As of **stoQ** version 0.10.3, plugin version checking is supported. If the min/max version of **stoQ** is not met, processing of the payload will proceed, but the user will be warned unpredictable results may be encountered.

Note: *Worker* plugins require the `hashpayload` and `saveresults` configuration options. No other plugins have additional requirements.

Note: *Worker* plugin supports a `max_tlp` option, which will limit it's ability to scan a payload based on the TLP level of the payload itself. Valid options are red, amber, green, and white. More information on TLP levels can be found at https://www.us-cert.gov/tlp

Note: *Worker* plugins support rate limiting. The value for `ratelimit` should be in the form of "count/per seconds". For example, the value `1/10` would mean **stoQ** will processes `1` sample every `10 seconds`.

2.3 Plugin Development

A *Worker* plugin extends the `StoqWorkerPlugin` class. As such, it must inherit the `StoqWorkerPlugin` class when initialized. In order to function properly, there must be several methods defined within the worker plugin.

- __init__
- activate

The `__init__` method is called upon initialization of the plugin. This occurs when the `Stoq.load_plugin` method is called with the plugin name or when `Stoq.collect_plugins` plugins is called.

The `activate` method is automatically called after the plugin has been initialized. When it is called, it must have `stoq` as an attribute. This allows the plugin to have full access to the **stoQ** framework and configuration options. The `activate` method should only be called once by the framework upon initialization. Any initial configuration and command line options should be placed here. This method must also return `True` in order for the framework to continue, otherwise **stoQ** will assume that the plugin activation has failed.

Additionally, the `deactivate` method is called when/if the plugin is ever deactivated, including when **stoQ** shuts down. This method is not required, though it is recommended should the plugin have any actions that need to cleaning up or if **stoQ** needs to deactivate the plugin for any reason.

For each of the above core methods, they should minimally call `super().METHOD_NAME()` right before they return. METHOD_NAME should be changed to the respective method. This will allow the respective parent class execute any required code.

For time-based events (periodic flushes of buffers, etc), every plugin can define a `wants_heartbeat` property of the plugin. If that property is True, then a separate thread will be launched by stoQ to call the plugin's `heartbeat` method. The `heartbeat` method will be called with the plugin object as its only argument (so `heartbeat` can be treated as a class method of the plugin). The `heartbeat` method will only be called once, and it is expected to loop to call whatever periodic actions the plugin wishes to take. For example

```
def heartbeat(self):
    while True:
        time.sleep(1)
        self._checkCommit()
```

Note: Actions performed in the heartbeat must be multithread/multiprocess safe. If the actions in the heartbeat may change the values of properties that other plugin methods (like `save`) may also change, it is the responsibility of the plugin to properly handle locking access to those objects, or find other methods of thread safety.

Note: Also, at present only Worker and Connector plugins are checked to see if they need heartbeats. Others may be added in the future if the need arises.

2.3.1 Workers

In addition to the above requirements, the below method is required for *Worker* plugins:

- scan

The `scan` method is called when command `stoq` command has a payload available for processing. `scan` requires two attributes, `payload` and `**kwargs`. `payload` is the payload that the plugin should process. If the plugin does not require a payload, `payload` will be `None`. `**kwargs` is a `dict` that contains the message provide by RabbitMQ, or some basic metadata if RabbitMQ is not utilized. Once the `scan` method has completed processing the payload, it should return it's results as a `dict` or `list`. If results are returned as a `list`, each item in the `list` will be processed separately by the `StoqConnectorPlugin`. This will result in multiple results being saved separately for each payload. This allows for worker plugins to save results without making multiple calls, such as when interacting with an API that returns multiple results or parsing an SMTP session that contains a stream of e-mails. Optionally, if the results do not need to be process, it can return `None`.

Below is an example of a basic worker plugin.

```
# Required imports
import argparse
from stoq.args import StoqArgs
from stoq.plugins import StoqWorkerPlugin


# The worker plugin class must be unique. It will be inheriting
# the StoqWorkerPlugin class.
class BasicWorker(StoqWorkerPlugin):

    def __init__(self):
        # In nearly all cases, we do not want to handle anything here
        super().__init__()

    # This function is required in order to initialize the worker.
    # The framework will call the activate() function upon initialization
    # and must return True in order for the framework to continue
    def activate(self, stoq):

        # Ensure the stoQ class is available throughout the
        # plugin
        self.stoq = stoq

        # Instantiate our workers command line argument parser
        parser = argparse.ArgumentParser()

        # Initialize the default requirements for a worker, if needed.
        parser = StoqArgs(parser)

        # Define the argparse group for this plugin
        worker_opts = parser.add_argument_group("Plugin Options")

        # Define the command line arguments for the worker
        worker_opts.add_argument("-r", "--rules",
                                 dest='rulepath',
                                 help="Path to rules file.")

        # The first command line argument is reserved for the framework.
        # The work should only parse everything after the first command
        # line argument. We must always use stoQ's argv object to ensure
        # the plugin is properly instantied whether it is imported or
        # used via a command line script
        options = parser.parse_args(self.stoq.argv[2:])

        # If we need to handle command line argument, let's pass them
        # to super().activate so they can be instantied within the worker
        super().activate(options=options)

        # Must return true, otherwise the framework believes something
        # went wrong
        return True

    # The framework will call the scan() function when it is ready to
    # scan. All of the initial functionality should reside here
    def scan(self, payload, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        # Must return a dict
        kwargs['err'] = "Need more to do!"
        return kwargs
```

Note: `super().activate(options=options)` must be called for the plugin to be fully initialized.

2.3.2 Connectors

In addition to the above requirements, the below methods are required for *Connector* plugins

- save

The `save` method is called to save a payload to the specified connector. It must have the `payload` and `**kwargs` attributes. The `payload` attribute should be the data that will be saved via the connector. `**kwargs` are any additional attributes that the method may require.

Optionally, the below methods can be provided.

- connect
- disconnect
- get_file

`connect` should be called when a connection, or reconnection, to the connector database is required. Ideally, logic should be placed in `save` that will call `connect` to verify a live connection still exists.

`disconnect` is called when the connector should cleanly disconnect from the database.

`get_file` is used if the database supports the saving of files. `get_file` may be used to retrieve any files that are saved to the connector. The `**kwargs` attribute should contain whatever datapoints are need to retrieve the file.

```
from stoq.plugins import StoqConnectorPlugin


class BasicConnector(StoqConnectorPlugin):

    def __init__(self):
        super().__init__()

    def activate(self, stoq):
        self.stoq = stoq

        # Any additonal requirements once the connector is activated
        # should be placed here

        super().activate()

    def get_file(self, **kwargs):

        # Code to retrieve file from this connector should be placed here

        # No results, carry on.
        return None

    def save(self, payload, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        """
        Save results to mongodb

        :param str payload: Content to be inserted into database
        :param dict **kwargs: Any additional attributes that should
                              be added to the GridFS object on insert
        """

        # Make sure we have a valid connection
        self.connect()

        # Code to handle saving of the results should be placed here

        super().save()

    def connect(self, force_connect=False):
        # Logic should reside here that determines if we have an
        # active/valid connection, and if not, make one. Otherwise
        # continue on so the framework can save it's results.
        super().connect()

    def disconnect(self):
        super().disconnect()
```

2.3.3 Readers

In addition to the above requirements, the below method is required for *Reader* plugins:

- read

The `read` method requires the `payload` attribute, and optionally `**kwargs`. The `payload` should be the content that the *Reader* plugin should process. Any additional attributes should be defined in `**kwargs`. Once the *Reader* plugin is done processing the `payload`, it should return its results.

```
from stoq.plugins import StoqReaderPlugin


class BasicReader(StoqReaderPlugin):

    def __init__(self):
        super().__init__()

    def activate(self, stoq):
        self.stoq = stoq
        super().activate()

    def read(self, payload, **kwargs):
        """
        Basic Reader

        :param bytes payload: Payload to be processed
        :returns: Content of payload

        """
        return payload
```

2.3.4 Sources

In addition to the above requirements, the below methods are required for *Source* plugins:

- ingest

The `ingest` method does not require any arrtributes when called. *Source* plugins should push data back to the worker by calling the `worker.multiprocess_put` method. This is will pull data back to the main method for processing data in and our of the framework to include retrieving payloads, hashing, metadata generation, result handling, and saving of results.

```
from stoq.plugins import StoqSourcePlugin


class FileSource(StoqSourcePlugin):

    def __init__(self):
        super().__init__()

    def activate(self, stoq):
        self.stoq = stoq
        super().activate()

    def ingest(self):

        path = "/tmp/bad.exe"
        self.stoq.worker.multiprocess_put(path=path, archive='file')

        return True
```

A source plugin also requires the `multiprocess` boolean configuration option in it's `.stoq` file under the [options] header. For example:

```
[options]
multiprocess = True
```

If set to `True`, the source plugin will be capable of being run with multiple instances simultaneously. Note: if `multiprocess` option is set to `False` the source will still be run in a Python process, but stoq will only run one instance of that process.

2.3.5 Extractors

In addition to the above requirements, the below methods are required for *Extractor* plugins:

- extract

`extract()` must be called with the `payload` parameter. Optionally, `**kwargs` may be provided. The plugin may return None or a list of tuples. Index 0 of the tuple must be a dict() containing metadata associated with the decoded content, and Index 1 must be the decoded content itself as bytes.

```
from stoq.plugins import StoqExtractorPlugin


class ExampleExtractor(StoqExtractorPlugin):

    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
    def activate(self, stoq):
        self.stoq = stoq
        super().activate()

    def extract(self, payload, **kwargs):

        # handle any extraction requirements here
        meta = {"size": len(payload), "type": "test"}
        return [(meta, payload)]
```

2.3.6 Carvers

In addition to the above requirements, the below methods are required for *Carver* plugins:

- carve

`carve()` must be called with the `payload` parameter. Optionally, `**kwargs` may be provided. The plugin may return None or a list of tuples. Index 0 of the tuple must be a dict() containing metadata associated with the decoded content, and Index 1 must be the decoded content itself as bytes.

```
from stoq.plugins import StoqCarverPlugin


class ExampleCarver(StoqExtractorPlugin):

    def __init__(self):
        super().__init__()

    def activate(self, stoq):
        self.stoq = stoq
        super().activate()

    def carve(self, payload, **kwargs):

        # handle any carving requirements here
        meta = {"size": len(payload), "type": "test"}
        return [(meta, payload)]
```

2.3.7 Decoders

In addition to the above requirements, the below methods are required for *Decoder* plugins:

- decode

`decode()` must be called with the `payload` parameter. Optionally, `**kwargs` may be provided. The plugin may return None or a list of tuples. Index 0 of the tuple must be a dict() containing metadata associated with the decoded content, and Index 1 must be the decoded content itself as bytes.

```
from stoq.plugins import StoqDecoderPlugin


class ExampleDecoder(StoqDecoderPlugin):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()

    def activate(self, stoq):
        self.stoq = stoq
        super().activate()

    def decode(self, payload, **kwargs):

        # handle any decoding requirements here
        meta = {"size": len(payload), "type": "test"}
        return [(meta, payload)]
```

2.3.8 Decorators

In addition to the above requirements, the below methods are required for *Decorator* plugins:

- decorate

`decorate()` must be called with the `results` parameter. The plugin *must* return a `dict` of the original results provided to it, or modified `results`.

Note: The `dict` returned from `decorate()` *WILL* be what is saved/returned from **stoQ**, so be extremely careful with how *results* is modified.

```
from stoq.plugins import StoqDecoratorPlugin


class ExampleDecorator(StoqDecoratorPlugin):

    def __init__(self):
        super().__init__()

    def activate(self, stoq):
        self.stoq = stoq
        super().activate()

    def decorate(self, results):
        # handle any logic to determine what is added to results here
        if 'APT' in results['scan']:
            results = {'apt_malware': True}
        return results
```

2.4 Packaging Plugins

stoQ provides a method to install plugins and their dependencies utilzing setuptool and pip. In order to leverage the plugin installation feature, some requirements must be met for the plugin package.

- The plugin package must be a directory
- The plugin directory must have a subdirectory by the same name as defined in the plugins `.stoq` configuration file

- The plugin directory must contain a valid **stoQ** configuration file
- The plugin directory must contain a valid **stoQ** plugin
- The plugin directory must contain a file named `__init__.py`
- Optionally, the archive/directory may contain a valid pip *requirements.txt* file. The pip packages within this file will automatically be installed with the **stoQ** plugin.
- Optionally, a MANIFEST.in file can be included to define which files within the package should be copied to the installation path.

Note: The plugin's configuration file will not be copied by default, this file should either be defined here or within `package_data` in `setup.py`.

As an example, a **stoQ** plugin archive should have the following structure:

```
basicworker-plugin/
    setup.py
    MANIFEST.in (optional)
    requirements.txt (optional)
    basicworker/
        __init__.py
        basicworker.stoq
        basicworker.py
```

The **stoQ** installation process will extract plugin options from it's `.stoq` configuration file. As such, the plugin's `setup.py` file should be fairly simple. The below `setup.py` should suffice for most plugins.:

```
from setuptools import setup, find_packages

setup(
    name=open("NAME").read(),
    version=open("VERSION").read(),
    author=open("AUTHOR").read(),
    url=open("WEBSITE").read(),
    license="Apache License 2.0",
    description=open("DESCRIPTION").read(),
    packages=find_packages(),
    include_package_data=True,
    classifiers=[
        "Development Status :: 3 - Alpha",
        "Topic :: Utilities",
    ],
)
```

2.4.1 Templates

stoQ allows for two types of outputs. First, a JSON blob that can be easily parsed in an automated fashion. In addition, **stoQ** can handle output using Jinja2 templating. This allows for highly customizable and human readable output that may be neccessary in many circumstances. As an example, when using the slack worker plugin, it is not ideal to have hundreds, maybe even thousands, of lines sent to a channel as a result of scanning a payload. With **stoQ**'s templating engine, one can easily send human readable and easily digested results to the Slack channel, while at the same time providing the JSON results to a connector for saving into a database for later use.

Using **stoQ**'s templates is a simple process. Simply create a `templates` directory in the plugin's directory and then create a new `template` file in Jinja2 format. For example, let's say we have a worker plugin by the name *peinfo*.

We want to create a Slack template for this plugin. All that is needed now is for a `slack.tpl` template to be placed in this directory. Now, we just need to run the slack worker with the `-T slack.tpl` argument. The slack worker plugin will then load the template and render the results.

Additionally, content that is passed to the connector plugin may also be parsed using the templating engine. In order to use this functionality, the worker plugin that is producing the data must have a template named after the connector plugin that is being utilized. For instance, if one would like to ensure the stdout connector output is human readable and not the JSON results, simply create a new template with the name `stdout.tpl` and call the worker with `-T stdout.tpl`.

2.4.2 Installing a Plugin

Installation of a **stoQ** plugin is very simple. Let's assume that we want to install the basicworker plugin that comes prepackaged with **stoQ**. We must first package the plugin as detailed above, and then run the command from within the **stoQ** directory:

```
stoq install basicworker-plugin


    .d8888b.  888             .d88888b.
   d88P  Y88b 888            d88P" "Y88b
   Y88b.      888            888     888
    "Y888b.   888888 .d88b.  888     888
       "Y88b. 888   d88""88b 888     888
         "888 888   888  888 888 Y8b 888
   Y88b  d88P Y88b. Y88..88P Y88b.Y8b88P
    "Y8888P"   "Y888 "Y88P"   "Y888888"
                                  Y8b

[+] Looking for plugin in /vagrant/stoq/plugin-packages/worker/yara...
[+] Installing yara plugin into /vagrant/stoq/stoq/plugins/worker...
[+] Install complete.
```

Let's examine what **stoQ** just did. First, we opened the *basicworker-plugin* plugin package and began searching for a **stoQ** plugin configuration file. Once it was found, we loaded it and searched for the Name and Module configuration options within the file. That allowed us to discover the plugin name along with the plugins .py filename. **stoQ** then discovered the plugin class to determine the full path where the plugin should be installed to. It then called pip to complete the installation.

If a file or directory exists, it will not be overwritten. Instead, a warning message will be displayed letting the user know that the plugin will not be installed. In order to successfully install the plugin, the file or directory must be removed, renamed, or –upgrade be called at the command line.

CHAPTER 3

Dispatcher

3.1 Overview

stoQ provides for the ability to dispatch, or route, payloads to other plugins. This is done by leveraging *yara* to identify payloads that have certain characteristics and then automatically routing to specific plugins based on the results. Currently two plugin categories are supported for use with dispatching, *extractor* and *carver*.

3.2 Usage

If dispatching is desired, simply start the worker with the `-D` command line argument. Ensure that your *dispatcher.yar* file contains the appropriate rules to properly route the payloads.

3.2.1 Writing a Dispatcher Rule

Dispatching relies on *yara* and a set of rules to appropriately route payloads to their intended plugin. As with any yara rules, the `strings` and `condition` parameters are required, but dispatching also requires the `meta` attribute. Two keys, `plugin` and `save` are required within the `meta` attribute. The `plugin` key identifies the **stoQ** plugin category and plugin name (e.g., `plugin = "carver:rtf"`) that should be loaded if the *yara* rule hits. It can contain multiple comma separated plugins that the payload should be dispatched to (e.g., `plugin = "carver:rtf, decoder:b64"`). There is no limit on how many plugins may be used for dispatching. The `save` key identifies whether content that is extracted or carved from the payload should be saved. Additionally, all of the meta values are passed to the specified plugin as **kwargs.

As an example, a **stoQ** dispatcher plugin that would identify RTF documents and then send the document to the RTF carver plugin would be written as:

```
rule rtf_file
{
    meta:
        plugin = "carver:rtf"
```

(continues on next page)

(continued from previous page)

```
        save = "True"
    strings:
        $rtf = "{\\rt" nocase
    condition:
        $rtf at 0
}
```

Results from the specified plugin are returned as a `list()` of `sets()`. Each unique object, or `payload`, that is extracted from the primary payload is assigned an incremental `payload` and a unique `uuid`. In version of **stoQ** prior to 0.9.38, a `puuid` key is also added to the results in order to identify the parent uuid the stream was extracted from. In **stoQ** version 0.9.38 and later, `uuid` is appended to a list for better tracking of parent child relationships. The results from the dispatcher are then appended to the primary results `dict()` and the key `payloads` is added with the total count of streams processed, to include the original payload.

3.3 Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Stoq

4.1 Overview

The *Stoq* class is the core of the framework. It must be instantiated in order for all other modules to function properly. This class is meant to be called from stoq.py.

Upon instantiation, default configuration options are defined within *__init__*. These are overridden if there is identical configuration option in *stoq.cfg*.

The *StoqPluginManager* will also be instantiated as a child class automatically. This allows for the ability to globally access the API for plugins and easily grant the ability for plugins to load other plugins.

4.2 Examples

Instantiate the *Stoq* class:

```
from stoq.core import Stoq
stoq = Stoq()
```

Retrieve a file from a url:

```
content = stoq.get_file("http://google.com")
```

Write content to disk:

```
stoq.write("example content", path="/tmp", filename="example.txt")
```

Note: If no filename is given, `Stoq.get_uuid` will be called and a random filename will be defined automatically. Additionally, if the filename already exists, the file will not be overwritten. However, if `Stoq.write()` is called with `overwrite=True`, the file will be overwritten. If the content to be written is binary, one may add `binary=True` when calling `Stoq.write()`.

In many cases, you may wish to define plugin options. This is especially so if you are not using *stoQ* from the command line. You may provide the parameter *plugin_options* when instantiating the *Stoq()* class.

Instantiate *Stoq* class, and set attributes for plugins:

```
from stoq.core import Stoq

plugin_options = {
    'worker': {
        'yara': {
            'yararules': '/data/yara/rules.yar'
            }
        }
    }
stoq = Stoq(plugin_options=plugin_options)
```

The plugin options will be available within the plugin object itself. For instance, in the above example the yara worker plugin will now have the attribute *yararules* defined as */data/yara/rules.yar*.

4.3 API

class stoq.core.**Stoq**(*argv=None*, *base_dir=None*, *log_dir=None*, *results_dir=None*, *temp_dir=None*, *plugin_dir_list=None*, *archive_base=None*, *config_file=None*, *dispatch_rules=None*, *useragent=None*, *plugin_options=None*, *log_level=None*, *log_maxbytes=None*, *log_backup_count=None*, *default_connector=None*, *default_source=None*, *filename_suffix=None*, *max_recursion=None*, *max_queue=None*, *source_base_tuple=None*, *url_prefix_tuple=None*, *log_syntax=None*, *sentry_url=None*, *sentry_ignore_list=None*, *default_tlp=None*)

Core stoQ Framework Class

dumps(*data*, *indent=4*, *compactly=False*)

Wrapper for json library. Dump dict to a json string

Parameters

- **data** (*dict*) – Python dict to convert to json
- **indent** (*int*) – Indent level for return value
- **compactly** – set to True to return unindented JSON (no newlines between key/values),

Returns Converted json string

Return type str

force_unicode(*payload*)

Force a string to be properly encoded in unicode using BeautifulSoup4

Parameters **payload** (*bytes*) – String to be forced into unicode

Returns Unicode bytes

Return type bytes

get_file(*source*, *params=None*, *verify=True*, *auth=None*, *timeout=30*, ***kwargs*)

Obtain contents of file from disk or URL.

Note: A file will only be opened from disk if the path of the file matches the regex defined by source_base_tuple in stoq.cfg.

Parameters

- **source** (*bytes*) – Path or URL of file to read.
- **params** (*bytes*) – Additional parameters to pass if requesting a URL
- **verify** (*bool*) – Ensure SSL Certification Verification
- **auth** – Authentication methods supported by python-requests
- **timeout** (*int*) – Time to wait for a server response
- ****kwargs** – Additional HTTP headers

Returns Content of file retrieved

Return type bytes or None

get_time
Get the current time, in ISO format

Returns Current time in ISO Format

Return type str

get_uuid
Generate a random uuid

Returns Random uuid

Return type str

hashpath(*sha1*)
Generate a path based on the first five chars of a SHA1 hash

example: The SHA1 4caa16eba080d3d4937b095fb68999f3dbabd99d would return a path similar to: /opt/malware/4/c/a/a/1

Parameters **sha1** (*str*) – SHA1 hash of a payload

Returns Path

Return type str

load_config()
Load configuration file. Defaults to stoq.cfg.

loads(*data*)
Wrapper for json library. Load json string as a python dict

Parameters **data** (*str*) – json string to load into dict

Returns Converted dict

Return type dict

logger_init()
Initialize the logger globally.

Returns True

normalize_json(*obj*)

Normalize json blobs:

- **If a key's value is a dict:**
  - Make the value a list
  - Iterate over sub keys and do the same
- **If a key's value is a list:**
  - Iterate over the values to ensure they are a string
- **If the key's value is anything else:**
  - Force the value to be a string

Parameters **obj** (*dict*) – dict object to normalize

Returns Normalized dict object

Return type dict

post_file(*url*, *params=None*, *files=None*, *data=None*, *auth=None*, *verify=True*, *timeout=30*, ***kwargs*)

Handles POST request to specified URL

Parameters

- **url** (*bytes*) – URL to for POST request
- **params** (*bytes*) – Additional parameters to pass if requesting a URL
- **files** (*tuple*) – Tuple of file data to POST
- **data** (*bytes*) – Content to POST
- **auth** – Authentication methods supported by python-requests
- **verify** (*bool*) – Ensure SSL Certification Verification
- **timeout** (*int*) – Time to wait for a server response
- ****kwargs** – Additional HTTP headers

Returns Content returned from POST request

Return type bytes or None

put_file(*url*, *params=None*, *data=None*, *auth=None*, *verify=True*, *timeout=30*, ***kwargs*)

Handles PUT request to specified URL

Parameters

- **url** (*bytes*) – URL to for PUT request
- **params** (*bytes*) – Additional parameters to pass if requesting a URL
- **data** (*bytes*) – Content to PUT
- **auth** – Authentication methods supported by python-requests
- **verify** (*bool*) – Ensure SSL Certification Verification
- **timeout** (*int*) – Time to wait for a server response
- ****kwargs** – Additional HTTP headers

Returns Content returned from PUT request

Return type bytes or None

sanitize_json(*obj*)
Sanitize json so keys do not contain '.' or ' '. Required for compaitibility with databases such as mongodb and elasticsearch

Parameters **obj** (*dict*) – dict object

Returns Sanitized dict object

Return type dict

write(*payload*, *filename=None*, *path=None*, *binary=False*, *overwrite=False*, *append=False*)
Write content to disk

Parameters

- **payload** (*str*) – Data to be written to disk
- **filename** (*str*) – Filename, if none is provided, a random filename will be used
- **path** (*str*) – Path for output file
- **binary** (*bool*) – Define whether content is binary or not
- **overwrite** (*bool*) – Define whether output file should be overwritten
- **append** (*bool*) – Define whether output file should be appended to

Returns Full path of file that was written

Return type str or False

CHAPTER 5

StoqArgs

5.1 Overview

StoqArgs() contains the primary command line arguments for the *stoQ* Framework. All command line options made available in this function will be made avilable to plugins that are extended with this function.

Note: Command line arguments defined within StoqArgs() will be made available globally within the *stoQ* Framework. Plugin command line arguments must not be defined here, but should instead be defined within the plugin itself.

5.2 Examples

From within a worker plugin, define command line arguments:

```
import sys
import argparse
from stoq.args import StoqArgs

# Instantiate our workers command line argument parser
parser = argparse.ArgumentParser()

# Initialize the default requirements for a worker, if needed.
parser = StoqArgs(parser)

# Define the argparse group for this plugin
worker_opts = parser.add_argument_group("Plugin Options")

# Define the command line arguments for the worker
worker_opts.add_option("-r", "--rules",
                       dest='rulepath',
```

(continues on next page)

(continued from previous page)

```
                        help="Path to rules file.")

# The first command line argument is reserved for the framework.
# The work should only parse everything after the first command
# line argument. We must always use stoQ's argv object to ensure
# the plugin is properly instantied whether it is imported or
# used via a command line script
options = parser.parse_args(self.stoq.argv[2:])

# If we need to handle command line argument, let's pass them
# to super().activate so they can be instantied within the worker
super().activate(options=options)
```

This will extend the command line arguments from those made available at initialization, to those defined in *worker_opts*. The variable `rulepath`, defined above, will be accessible by calling `worker.rulepath`

5.3 API

`stoq.args.`**`StoqArgs`**(*parser*)

Initializes command line arguments within the plugin

Parameters **`parser`** – argparse object for parsing

Returns Modified argparse object

CHAPTER 6

StoqPluginManager

6.1 Overview

StoqPluginManager() is the primary class that controls all aspects of plugin management to include initialization, loading, listing, and unloading. This class is instantiated within the *Stoq()* class. This should not be instantiated outside of **stoQ** as it relies on objects within *Stoq()* to function properly.

Note: Full plugin development documentation can be found at *Plugin Development*.

6.2 Examples

Instantiate Stoq:

```
from stoq.core import Stoq
stoq = Stoq()
```

Listing all available plugins:

```
stoq.list_plugins()
```

Once Stoq() is initialized, we can load a worker. The worker should always be instantiated first, then any additional plugins may be loaded through the worker plugin itself. The plugins will be instantiated within a dict in the worker plugin class. For example, a **stoQ** connector plugin may be accessed from it's plural name (connectors) within the worker object by calling `worker.connectors` or a reader plugin may be called with `worker.readers`:

```
worker = stoq.load_plugin("yara", "worker")
worker.load_connector("file")
payload = worker.connectors['file'].get_file(path="/tmp/bad.exe")
results = worker.scan(payload)
```

We may also retrieve a payload from a connector, such as MongoDB:

```
worker.load_connector("mongodb")
file_hash = "da39a3ee5e6b4b0d3255bfef95601890afd80709"
payload = worker.connectors['mongodb'].get_file(sha1=file_hash)
results = worker.scan(payload)
```

Note: Only certain connector plugins support `.get_file(**kwargs)`. Refer to the plugin to determine if it is supported or not.

Now that we have results, we can load our connector to save the results:

```
worker.connectors['mongodb'].save(results)
```

We may also save a file via the connector. In this example, we will save a payload to with some additional attributes to GridFS:

```
payload_attributes = {}
payload_attributes['md5'] = "d41d8cd98f00b204e9800998ecf8427e"
payload_attributes['sha1'] = "da39a3ee5e6b4b0d3255bfef95601890afd80709"
payload_attributes['sha256'] =
↪"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
worker.connectors['mongodb'].save(payload, archive=True, payload_attributes)
```

Note: `save()` accepts `**kwargs`, so one may pass any attribute that is needed to it. GridFS will automatically calculate the payload size and datetime uploaded.

6.3 API

class stoq.plugins.**StoqPluginManager**

stoQ Plugin Manager Class

collect_plugins()

Find all stoQ plugins and their configuration file

get_all_plugin_names

List all plugin names

Returns All plugin names

Return type list

get_all_plugins

List all valid plugins and configurations

Returns All valid plugins

Return type dict

get_categories

Create list of plugin categories available

get_plugin(*name*, *category*)

Initializes a plugin within a specific category

Parameters

- **name** (*str*) – Name of plugin to get
- **category** (*str*) – Category of the named plugin

Returns plugin object

Return type object

get_plugins_of_category(*category*)
Lists plugin name of a specific category

Parameters **category** (*str*) – Category to discover plugins in

Returns A tuple of discovered plugins

Return type tuple

list_plugins()
List all available plugins and their category

load_plugin(*name*, *category*)
Load the desired plugin

Parameters

- **name** (*str*) – Plugin name to be loaded
- **category** (*str*) – The category of plugin to be loaded

Returns The loaded plugin object

Return type object

CHAPTER 7

StoqScan

7.1 Overview

Basic scanning functions such as hash calculation and file type detection.

7.2 Examples

Calculate the md5 hash of a payload:

```
import stoq.scan
stoq.scan.get_md5("this is a payload")
```

Calculate the md5, sha1, sha256, and sha512 of a payload:

```
stoq.scan.get_hashes("this is a payload")
```

7.3 API

stoq.scan.**bytes_frequency**(*payload*, *min_length=1*, *max_length=3*, *min_count=10*)

Determine the frequency of bytes or series of bytes in a payload

Parameters

- **payload** (*bytes*) – Payload to be analyzed
- **min_length** (*int*) – Minimum length of continuous bytes
- **max_length** (*int*) – Maximum length of continuous bytes
- **min_count** (*int*) – Minimum count of instances of a specific byte or series of bytes

Returns Bytes, count, percentage of frequency

Return type tuple

stoq.scan.**compare_ssdeep**(*payload1*, *payload2*)
Compare binary payloads with ssdeep to determine

Parameters

- **payload1** (*bytes*) – Binary content to compare
- **payload2** (*bytes*) – Binary content to compare

Returns Match score from 0 (no match) to 100

Type int or None

stoq.scan.**get_hashes**(*payload*)
Calculate the md5, sha1, sha256, and sha512 of a payload

Parameters **payload** – The payload to be hashed.

Returns All of the above hashes

Return type dict

stoq.scan.**get_magic**(*payload*, *mime=True*)
Attempt to identify the magic of a payload

Parameters

- **payload** (*bytes*) – Payload to be analyzed
- **mime** (*bool*) – Define whether the payload is of mime magic_type

Returns Identified magic type, otherwise None

Return type bytes

stoq.scan.**get_md5**(*payload*)
Generate md5 hash of a payload

Parameters **payload** – The payload to be hashed.

Returns md5 hash

Return type str

stoq.scan.**get_sha1**(*payload*)
Generate sha1 hash of a payload

Parameters **payload** – The payload to be hashed.

Returns sha1 hash

Return type str

stoq.scan.**get_sha256**(*payload*)
Generate sha256 hash of a payload

Parameters **payload** – The payload to be hashed.

Returns sha256 hash

Return type str

stoq.scan.**get_sha512**(*payload*)
Generate sha512 hash of a payload

Parameters **payload** – The payload to be hashed.

Returns sha512 hash

Return type str

stoq.scan.**get_ssdeep**(*payload*)
Generate ssdeep hash of a payload

Parameters **payload** – The payload to be hashed.

Returns ssdeep hash

Return type str or None

CHAPTER 8

StoqBloomFilter

8.1 Overview

Native support for bloom filters.

8.2 Examples

Create new bloom filter with a maximum of 5000 items and a false positive ratelimited of 0.0001%:

```
from stoq.filters import StoqBloomFilter

bloomfilter = StoqBloomFilter()
bloomfilter.create_filter("/tmp/stoq.bloom", 5000, 0.001)
```

Open a previously created bloom filter:

```
from stoq.filters import StoqBloomFilter

bloomfilter = StoqBloomFilter()
bloomfilter.import_filter("/tmp/stoq.bloom")
```

Save the bloomfilter to disk every 60 seconds:

```
bloomfilter.backup_scheduler(60)
```

Check if a string is in the bloom filter, if not, add it:

```
bloomfilter.query_filter("google.com", add_missing=True)
```

class stoq.filters.**StoqBloomFilter**

backup_scheduler(*interval*)
Set a syncing schedule for the persistent bloom filter

Parameters **interval** (*int*) – Interval between syncing bloom filter to disk

create_filter(*filepath*, *size*, *falsepos_rate*)
Create new bloom filter

Parameters

- **filepath** (*bytes*) – Path to persistent bloom filter on disk
- **size** (*int*) – Maximum number of elements in bloom filter
- **falsepos_rate** (*float*) – Maximum false positive probability

import_filter(*filepath*)
Load a previously created persistent bloom filter

Parameters **filepath** (*bytes*) – Path to persistent bloom filter on disk

query_filter(*item*, *add_missing=False*)
Identify whether an item exists within filter or not

Parameters

- **item** (*bytes*) – Item to query the bloom filter with
- **add_missing** (*bool*) – If set to True, the item will be added to the bloom filter if it doesn't exist

Returns True if item exists, False if not.

Return type bool

CHAPTER 9

StoqShell

9.1 Overview

A stoQ Interactive Shell

9.2 Examples

Instantiate a stoQ Interactive Shell session:

```
$ stoq shell

 .d8888b.  888              .d88888b.
d88P  Y88b 888             d88P" "Y88b
Y88b.      888             888     888
 "Y888b.   888888 .d88b.   888     888
    "Y88b. 888   d88""88b 888     888
      "888 888   888  888 888 Y8b 888
Y88b  d88P Y88b. Y88..88P Y88b.Y8b88P
 "Y8888P"   "Y888 "Y88P"   "Y888888"
                                Y8b
        Analysis. Simplified.


[stoQ] >
```

List all available plugins:

```
[stoQ] > list
Available Plugins:
connectors
- stdout              v0.9    Sends content to STDOUT
- file                v0.9    Retrieves and saves content to local disk
```

(continues on next page)

(continued from previous page)

```
extractors
- decompress          v0.9    Extract content from a multitude of archive formats
- gpg                 v0.1    Handle GnuPG encrypted content
carvers
- pe                  v0.9    Carve portable executable files from a data stream
- swf                 v0.9    Carve and decompress SWF payloads
- ole                 v0.9    Carve OLE streams within Microsoft Office Documents
- xdp                 v0.9    Carve and decode streams from XDP documents
- rtf                 v0.9    Carve hex/binary streams from RTF payloads
readers
- pdftext             v0.9    Extract text from a PDF document
- tika                v0.1    Upload content to a Tika server for automated text␣
↪extraction
- iocregex            v0.9    Regex routines to extract and normalize IOC's from a␣
↪payload
sources
- rabbitmq            v0.9    Publish and Consume messages from a RabbitMQ Server
- dirmon              v0.9    Monitor a directory for newly created files for␣
↪processing
- filedir             v0.9    Ingest a file or directory for processing
workers
- peinfo              v0.9    Gather relevant information about an executable using␣
↪pefile
- exif                v0.9    Processes a payload using ExifTool
- publisher           v0.9    Publish messages to single or multiple RabbitMQ queues␣
↪for processing
- trid                v0.4    Identify file types from their TrID signature
- xorsearch           v0.9    Search a payload for XOR'd strings
- yara                v0.9    Process a payload using yara
- iocextract          v0.9    Utilizes reader/iocregex plugin to extract indicators␣
↪of compromise from documents
decoders
- rot47               v0.1    Decode ROT47 encoded content
- bitwise_rotate      v0.1    Rotate bits left or right. Defaults to 4 bits right for␣
↪nibble swapping.
- b64                 v0.1    Decode base64 encoded content
- b85                 v0.1    Decode base85 encoded content
- xor                 v0.1    Decode XOR encoded content
```

Load the yara plugin:

```
[stoQ] > load worker yara
```

Conduct a simple scan of a payload using only the yara plugin:

```
[stoQ] > read /tmp/bad.exe
[*] Read /tmp/bad.exe(510968 bytes)
[*] sha1: 074c5b3707ebcda408a186082e529cf8ae5859ed
[*] sha256: 3cb2eb909ea3cfac42621ed4d024ed9d15a2005cc91a54050ef75fc9bee695b7
[*] sha512:␣
↪53fcb7f9087b5f356067f6f2cd288575e97876fdad9e1376231923e414b541b0fdba7f17095daba0899155f2cde11efb5d4
[*] md5: 0b40e4e5987e7fb14b7a9b9b9218c703
[*] magic: application/x-dosexec
[stoQ] > run worker yara
[stoQ] > results
{ "hits" : [ {
        "matches" : true,
```

(continues on next page)

(continued from previous page)

```
        "meta" : {
                "author" : "PUNCH Cyber Analytics Group",
                "cve" : "N/A",
                "description" : "Badness",
                "type" : "Suspicious String",
                "version" : "1.0",
                "weight" : 100
                },
        "namespace" : "default",
        "rule" : "win_api_LoadLibrary",
        "strings" : [
                [
                    "23967",
                    "$LoadLibrary",
                    "b'LoadLibrary'"
                ],
            ],
        "tags" : [  ]
        } ],
    }
```

Display all available settings:

```
[stoQ] > set
worker.yara.saveresults = True
worker.yara.max_processes = 1
worker.yara.website = https://github.com/PUNCH-Cyber/stoq-plugins-public
worker.yara.templates = plugins/worker/yara/templates/
worker.yara.carvers = {}
worker.yara.template = False
worker.yara.readers = {}
worker.yara.plugin_path = /usr/local/stoq/plugins/worker/yara
worker.yara.dispatch = False
worker.yara.version = 0.9
worker.yara.description = Process a payload using yara
worker.yara.yararules = plugins/worker/yara/rules/stoq.yar
worker.yara.name = yara
worker.yara.path = False
worker.yara.module = /usr/local/stoq/plugins/worker/yara/yarascan
worker.yara.extractors = {}
worker.yara.archive_connector = False
worker.yara.source_plugin = False
worker.yara.workers = {}
worker.yara.decoders = {}
worker.yara.category = worker
worker.yara.log_level = False
worker.yara.hashpayload = True
worker.yara.is_activated = True
worker.yara.output_connector = stdout
worker.yara.author = Marcus LaFerrera
worker.yara.error_queue = False
worker.yara.sources = {}
stoq.config_file = /usr/local/stoq/stoq.cfg
stoq.default_connector = stdout
stoq.log_dir = /usr/local/stoq/logs
stoq.log_maxbytes = 1500000
stoq.log_path = /usr/local/stoq/logs/stoq.log
```

(continues on next page)

(continued from previous page)

```
stoq.base_dir = /usr/local/stoq
stoq.useragent = Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1)
stoq.url_prefix_tuple = http://, https://
stoq.results_dir = /usr/local/stoq/results
stoq.temp_dir = /usr/local/stoq/temp
stoq.dispatch_rules = /usr/local/stoq/dispatcher.yar
stoq.default_source = filedir
stoq.log_level = INFO
stoq.log_backup_count = 5
stoq.source_base_tuple = /usr/local/stoq
stoq.max_recursion = 3
stoq.plugin_dir_list = /usr/local/stoq/plugins
stoq.archive_base = /usr/local/stoq/archive
```

Update a configuration setting:

```
[stoQ] > set stoq.log_level DEBUG
stoq.log_level -> DEBUG
```

Save results, to include any payloads that may have been carved/extracted/decoded. If mutiple results have been processed, the integer will be incremented and correspond to the payload id viewable in the `results` command:

```
[stoQ] > save
[*] Saving content to /usr/local/stoq/results/results-0-bad.exe
```

Now's let pass arguments to a plugin. In this instance we want to XOR a payload using a specific XOR key:

```
[stoQ] > run decoder xor key=2
[*] Run using xor complete. View results with 'results'
```

List contents of a directory:

```
[stoQ] > ls /tmp
bad.exe
```

class stoq.shell.**StoqShell**(*stoq*)

do_EOF(*input*)

do_exit(*input*)

do_list(*input*)

list List available plugins

do_load(*input*)

load <category> <plugin> Load plugin of category

do_ls(*input*)

ls <path> List contents in the specified directory

do_payload(*input*)

payload <id> Switch object to scan to an extracted stream

do_read(*input*)

read <path to file> Open a file at specified path

do_results (*input*)

results Display results of previous plugin run

do_run (*input*)

run <category> <plugin> [key=value] Run an individual plugin against the loaded payload

do_save (*input*)

save [payload | id] Save all results, the current payload, or only a specific results ID to disk

do_set (*input*)

set <global setting> <new value> Set global setting to value

do_usage (*input*)

usage <category> <plugin> Display any documentation available for the specified plugin

set_prompt (*msg=''*)

CHAPTER 10

Overview

stoQ is a automation framework that helps to simplify the more mundane and repetitive tasks an analyst is required to do. It allows analysts and DevSecOps teams the ability to quickly transition from different data sources, databases, decoders/encoders, and numerous other tasks. stoQ was designed to be enterprise ready and scalable, while also being lean enough for individual security researchers.

CHAPTER 11

Usage

stoQ can be run in several modes to include interactive shell, single file, entire directories, monitoring directories for new files, and queue mode. Let's go over some of the simplest ways of using **stoQ**.

11.1 Basic Usage via Interactive Shell

stoQ provides a simply interactive shell interface. This interface is designed to allow a user to interact with **stoQ** and **stoQ** plugins on a much more granular level than via the command line.

To enter the interactive shell, simply run **stoQ** with the `shell` argument.:

```
bash$ stoq shell


    ________ __________ _________  ________
    (  _____ \__    __/(  ____  )(  ____  )
    | (     \/   ) (   | (    ) || (    ) |
    | (_____     | |   | |    | || |    | |
    (_____  )    | |   | |    | || |    | |
          ) |    | |   | |    | || | /\| |
    /\____) |    | |   | (____) || (_\ \ |
    \________)    )_(   (________)(_____\/_)

             Analysis. Simplified.
                   v0.9.7

[stoQ] >
```

Once in the interactive shell, you can run the `help` command for a complete listing of available commands. Please view the *StoqShell* documentation for a more exhaustive list of directions.

11.2 Basic Usage via Command Line

In order to use **stoQ** via the command line, at least two options must be defined. The worker plugin that should be loaded, and the source of input. In order to see a basic usage help, simply execute `stoq`:

```
bash$ stoq

    .------..------..------..------.
    |S.--. ||T.--. ||O.--. ||Q.--. |
    | :/\: || :/\: || :/\: || (\/) |
    | :\/: || (__) || :\/: || :\/: |
    | '--'S|| '--'T|| '--'O|| '--'Q|
    `------'`------'`------'`------'
          Analysis. Simplified.
                v0.9.7

usage:
    stoq [command] [<args>]

    Available Commands:
        help    Display help message
        shell   Launch an interactive shell
        list    List plugins available
        worker  Load specified worker plugin
        install Install a stoQ plugin
```

To view a complete listing of available plugins simply call `stoq` with the `list` command line argument:

```
bash$ stoq list

    _______ ________  ______   ______
    |_______     |    |      | |    __|
    _______|     |    |______| |____\|

         Analysis. Simplified.
               v0.9.7

Available Plugins:
 connectors
    - s3                  v0.1    Sends and retrieves content from Amazon S3 buckets
    - queue               v0.1    Send results to a queuing system, such as RabbitMQ
    - mongodb             v0.9    Sends and retrieves content from MongoDB
    - emailer             v0.1    Send results to recipients via e-mail
    - elasticsearch       v0.2    Saves content to an ElasticSearch index
    - stdout              v0.9    Sends content to STDOUT
    - file                v0.9    Retrieves and saves content to local disk
    - fluentd             v0.1    Sends content to a fluentd server
 sources
    - rabbitmq            v0.9    Publish and Consume messages from a RabbitMQ Server
    - dirmon              v0.9    Monitor a directory for newly created files for␣
↪processing
    - filedir             v0.9    Ingest a file or directory for processing
 carvers
    - pe                  v0.9    Carve portable executable files from a data stream
    - swf                 v0.9    Carve and decompress SWF payloads
    - ole                 v0.9    Carve OLE streams within Microsoft Office Documents
    - xdp                 v0.9    Carve and decode streams from XDP documents
```

(continues on next page)

(continued from previous page)

```
    - rtf                 v0.9    Carve hex/binary streams from RTF payloads
 workers
    - basicworker         v0.1    StoQ framework example of a basic worker plugin
    - peinfo              v0.9    Gather relevant information about an executable␣
↪using pefile
    - passivetotal        v0.5    Query PassiveTotal API for a domain or IP address
    - threatcrowd         v0.1    Interact with ThreatCrowd API
    - opswat              v0.9    Submit content to an OPSWAT Metascan server for␣
↪scanning and retrieve the results
    - exif                v0.9    Processes a payload using ExifTool
    - publisher           v0.9    Publish messages to single or multiple RabbitMQ␣
↪queues for processing
    - trid                v0.4    Identify file types from their TrID signature
    - totalhash           v0.7    Query TotalHash API for analysis results
    - xorsearch           v0.9    Search a payload for XOR'd strings
    - clamav              v0.1    Scan content with ClamAV
    - yara                v0.9    Process a payload using yara
    - censys              v0.2    Interact with Censys.io API
    - iocextract          v0.9    Utilizes reader/iocregex plugin to extract␣
↪indicators of compromise from documents
    - vtmis               v0.9    Interact with VTMIS public and private API
    - slack               v0.9    Interact with StoQ Plugins using Slack as an␣
↪interface
    - fireeye             v0.1    Saves a file into a directory fireeye monitors via␣
↪CIFS for analysis
 readers
    - pdftext             v0.9    Extract text from a PDF document
    - tika                v0.1    Upload content to a Tika server for automated text␣
↪extraction
    - iocregex            v0.9    Regex routines to extract and normalize IOC's from␣
↪a payload
 extractors
    - decompress          v0.9    Extract content from a multitude of archive formats
    - gpg                 v0.1    Handle GnuPG encrypted content
 decoders
    - rot47               v0.1    Decode ROT47 encoded content
    - bitrot              v0.1    Rotate bits left or right. Defaults to 4 bits right␣
↪for nibble swapping.
    - b64                 v0.1    Decode base64 encoded content
    - b85                 v0.1    Decode base85 encoded content
    - xor                 v0.1    Decode XOR encoded content
```

Now that we have a complete listing of available worker and connector plugins, we can begin processing data. Let's say that we have a file named *bad.exe* that we want to process with the *yara* worker plugin. We also want the results to be displayed to our console. We can simply run **stoQ** with the following command line arguments:

```
bash$ stoq yara -F bad.exe
{
"date" : "2015-10-29T15:22:55.824563",
"payloads" : 1,
"results" : [ {
        "md5" : "0ace1c67d408986ca60cd52272dc8d35",
        "payload_id" : 0,
        "plugin" : "yara",
        "scan" : [ { "matches" : true,
                     "meta" : {
```

(continues on next page)

(continued from previous page)

```
                                "author" : "PUNCH Cyber Analytics Group",
                                "cve" : "N/A",
                                "description" : "Badness",
                                "type" : "Suspicious String",
                                "version" : "1.0",
                                "weight" : 100
                                },
                            "namespace" : "default",
                            "rule" : "win_api_LoadLibrary",
                            "strings" : [
                                    [
                                        "23967",
                                        "$LoadLibrary",
                                        "b'LoadLibrary'"
                                    ],
                                ],
                            "tags" : [  ]
                        }
                    ],
            "sha1" : "5a04547c1c56064855c3c6426448d67ccc1e0829",
            "sha256" : "458f1bb61b7ef167467228141ad44295f3425fbeb6303e9d31607097d6869932",
            "sha512" :
↪"c5dbd244d186546846c25a393edeafdd6604e2a2e04e021a21d0524f7b02d3ecb85c12dba252a11a3bb01c20fb736ca615
↪",
            "size" : 55208,
            "uuid" : ["da8215ed-89ca-43db-8c96-a8b8231f6a5e"]
        } ]
}
```

We can easily change the method the results are handled by modifying the `-C` flag. Simply replace `stdout` with another plugin name, such as `file` or `mongodb`. The default *connector* plugin may also be changed by changing the `output_connector` option in *stoq.cfg*.

Additionally, output can be customized using **stoQ**'s templating engine.

11.3 Using the queues

Queues enable **stoQ** to process payloads in a ditributed and scalable manner. In this use case, we will utilize the *publisher* worker plugin with RabbitMQ. The *publisher* worker plugin's primary purpose is to handle files to be ingested, and then notify the other worker plugins that there is a file that is ready to be processed. By default, the *publisher* worker plugin will notify each of the worker plugins that are defined in *publisher.stoq*. This can be easily modified at run time by defining one or many `-w` command line arguments for the *publisher*. For now, we will assume that the default worker queues (*yara, exif, peinfo, trid*) are sufficient.

Let's assume that we have a directory in our current working directory named *malicious*. We want to monitor this directory, using the dirmon source plugin, for any new files that are created, archive them to MongoDB, and then process them with our default workers listed above:

```
bash$ stoq publisher -I dirmon -F malicious -A mongodb
```

Once a file is placed into this directory, the newly created file will be ingested, saved into our MongoDB instance, and a message will be sent to the appropriate queues for processing.

Now, we need to make sure our worker plugins are running so they can processes their newly identified file. In this scenario, since we are saving the file itself into MongoDB, we will also save our worker plugin results into MongoDB:

```
bash$ stoq yara -I rabbitmq -C mongodb &
bash$ stoq exif -I rabbitmq -C mongodb &
bash$ stoq peinfo -I rabbitmq -C mongodb &
bash$ stoq trid -I rabbitmq -C mongodb &
```

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Q

S

W