stoQ Documentation

Release v3.0.1

Marcus LaFerrera

Jan 06, 2022

CONTENTS

Release v3.0.1

CHAPTER

ONE

OVERVIEW

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

1.1 History

stoQ was initially a collection of scripts that helped us solve problems we encountered daily. These tasks, such as parsing an SMTP session, extracting attachments, scanning them with a multitude of custom and open source tools, saving the results, and then finally analyzing them took up an increasing amount of our team's resources. We spent an ever increasing amount of time simply attempting to collect and extract data. This took valuable resources away from our ability to actually find and analyze adversaries targeting our networks.

We grew tired of being the hamster in a wheel and decided to do something about it. In 2011, we began development of a framework that would not only tackle the problem above, but also allow us to quickly change the flow of data and automated analytics, quickly pivot to new databases to house the results, and simply be able to respond to the adversaries changing their tactics, techniques, and procedures (TTPs).

Most importantly, our focus was to build a tool that would allow us to do what we love to do – defend networks from adversaries that are determined, focused, and relentless.

In 2015, after stoQ had been matured in multiple large scale operational networks, we decided to open source our work in hopes of helping the wider Network Defense community. Since then, we've been constantly enhancing stoQ thanks to the feedback and contributions from the community of stoQ users.

1.2 Why use stoQ?

Over the years, there have been several other open source solutions that have been released that have similar capabilities to stoQ. However, stoQ is fundamentally different in many ways when compared to other solutions available. Some key differences are:

- Extremely lightweight and designed with simplicity in mind.
- Fully supports AsyncIO
- A wide range of publicly available plugins.
- *stoQ* makes no assumptions about your workflow. Analysts decide everything, from where data originates, how it is scanned/decoded/processed, to where it is saved.

- Scalable in not only native/bare metal environments, but also using solutions such as Kubernetes, AWS Lambda, Google Cloud Functions, Azure Functions, and many more.
- Written to be easily and quickly extended. All you need is a plugin.
- Can be used in an enterprise environment or by individuals without the need for client/server infrastructure
- Over 95% of code is covered by unittests.
- All core functions and plugins leverage typing and are type-checked at commit.
- Actively developed since 2011, open source since 2015.
- Extensive up-to-date documentation.

1.3 Philosophy

Our goal with stoQ has always been to simplify the mundane and automate the repetitive, ultimately enabling network defenders to do what they do best – focus on the threats. Since we began development, this philosophy has not shifted. Our core philosophy for both design and development can be best summarized by the Zen of Python:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Readability counts.
```

1.4 Architecture

One of the most powerful features in stoQ is its flexibility. Because stoQ is a framework, the majority of the work actually happens within the plugins. stoQ itself is meant to orchestrate the communication between the various plugins and normalize their results. stoQ makes no assumptions on the architecture that works best for the user. Because of this, stoQ allows for a highly configurable and flexible architecture that can be defined by the user.

For example, analysts can run stoQ against an individual file on their local computer, or against 100's of millions of payloads that are extracted off the wire – and everything in between. Payloads can be dynamically routed to plugins using yara, TRiD, and even static attributes. Results can be saved with ElasticSearch one day, then in Splunk the next, or both at the same time. Directories can be monitored for new files, queueing solutions such as RabbitMQ or Google PubSub can be leveraged, or mailboxes can even be monitored for new e-mails. No matter what an analyst wants to do with stoQ, it's simply a matter of writing a plugin.

1.5 Example Output

As an example of output from stoQ, let's scan a local file with ExifTool and get the hashes of the payload:

```
{
    "time": "...",
    "results": [
        {
            "payload_id": "00d2f069-d716-43ed-bc2f-b0bd295574d4",
            "size": 507904,
```

(continues on next page)

(continued from previous page)

```
            "payload_meta": {
                "should_archive": true,
                "extra_data": {
                    "filename": "bad.exe"
                },
                "dispatch_to": []
            },
            "workers": {
                "hash": {
                    "sha256":
↪"47c6e9b102324ea6c54dd95ad3fdf4b48b18775053b105e241a371a3731488c0",
                    "md5": "16d9f6e5491d99beb46d7ab1500c1799",
                    "sha1": "9e6414bf2802c98fbd13172817db80380c5eeb6a"
                },
                "exif": {
                    "SourceFile": "/tmp/tmp3r4juo8e",
                    "ExifToolVersion": 11.11,
                    "FileName": "tmp3r4juo8e",
                    "Directory": "/tmp",
                    "FileSize": 507904,
                    "FileModifyDate": "...",
                    "FileAccessDate": "...",
                    "FileInodeChangeDate": ".",
                    "FilePermissions": 600,
                    "FileType": "Win32 EXE",
                    "FileTypeExtension": "EXE",
                    "MIMEType": "application/octet-stream",
                    "MachineType": 332,
                    "TimeStamp": "2013:04:20 10:50:10-04:00",
                    "ImageFileCharacteristics": 258,
                    "PEType": 267,
                    "LinkerVersion": 9.0,
                    "CodeSize": 386048,
                    "InitializedDataSize": 120832,
                    "UninitializedDataSize": 0,
                    "EntryPoint": 208320,
                    "OSVersion": 5.0,
                    "ImageVersion": 0.0,
                    "SubsystemVersion": 5.0,
                    "Subsystem": 2
                }
            }
            "archivers": {},
            "plugins_run": {
                "workers": [
                    [
                        "exif",
                        "hash"
                    ]
                ],
                "archivers": []
            },
            "extracted_from": null,
            "extracted_by": null
        }
    ],
    "request_meta": {
```

(continues on next page)

(continued from previous page)

```
        "archive_payloads": true,
        "source": null,
        "extra_data": {}
    },
    "errors": {},
    "decorators": {},
    "scan_id": "4d053d5e-9f4e-417b-8f0e-deea0d45449d"
}
```

Or, carve a few executable files out of a Microsoft Word document:

```
{
    "time": "...",
    "results": [
        {
            "payload_id": "e777051a-832b-489f-b74c-9949b2c9a2ce",
            "size": 558592,
            "payload_meta": {
                "should_archive": true,
                "extra_data": {
                    "filename": "sample_doc_with_pe.doc"
                },
                "dispatch_to": []
            },
            "workers": {
                "exif": {
                    "SourceFile": "/tmp/tmpbqtisxjd",
                    "ExifToolVersion": 11.11,
                    "FileName": "tmpbqtisxjd",
                    "Directory": "/tmp",
                    "FileSize": 558592,
                    "FileModifyDate": "...",
                    "FileAccessDate": "...",
                    "FileInodeChangeDate": "...",
                    "FilePermissions": 600,
                    "FileType": "DOC",
                    "FileTypeExtension": "DOC",
                    "MIMEType": "application/msword",
                    "Identification": 42476,
                    "LanguageCode": 1033,
                    "DocFlags": 4616,
                    "System": 0,
                    "Word97": 0,
                    "Author": "xxxxxxxxxxxx",
                    "Template": "Normal",
                    "LastModifiedBy": "xxxxxxxxxxxx",
                    "Software": "Microsoft Office Word",
                    "CreateDate": "2017:11:13 21:27:00",
                    "ModifyDate": "2017:11:13 21:28:00",
                    "Security": 0,
                    "CodePage": 1252,
                    "Company": "",
                    "CharCountWithSpaces": 20,
                    "AppVersion": 14.0,
                    "ScaleCrop": 0,
                    "LinksUpToDate": 0,
                    "SharedDoc": 0,
```

(continues on next page)

(continued from previous page)

```
                    "HyperlinksChanged": 0,
                    "TitleOfParts": "",
                    "HeadingPairs": [
                        "Título",
                        1
                    ],
                    "CompObjUserTypeLen": 36,
                    "CompObjUserType": "Documento do Microsoft Word 97-2003",
                    "LastPrinted": "0000:00:00 00:00:00",
                    "RevisionNumber": 2,
                    "TotalEditTime": 1,
                    "Words": 3,
                    "Characters": 18,
                    "Pages": 1,
                    "Paragraphs": 1,
                    "Lines": 1
                },
                "hash": {
                    "sha256":
↪"4e3a682b2187f7c722b88af9bff5292fd7beb4d77233d1b3bc46f0bfc4891068",
                    "md5": "137720063880f80270a61181b021d000",
                    "sha1": "08bc0a52ee27ad0ceaa87bf394b1faa7a43bf27e"
                }
            }
            "archivers": {},
            "plugins_run": {
                "workers": [
                    [
                        "pecarve",
                        "exif",
                        "hash"
                    ]
                ],
                "archivers": []
            },
            "extracted_from": null,
            "extracted_by": null
        },
        {
            "payload_id": "471b49f3-ea99-481f-a0a3-502826e69c73",
            "size": 31232,
            "payload_meta": {
                "should_archive": true,
                "extra_data": {
                    "offset": 11367
                },
                "dispatch_to": []
            },
            "workers": {
                "exif": {
                    "SourceFile": "/tmp/tmpyi0yx_wf",
                    "ExifToolVersion": 11.11,
                    "FileName": "tmpyi0yx_wf",
                    "Directory": "/tmp",
                    "FileSize": 31232,
                    "FileModifyDate": "...",
                    "FileAccessDate": "...",
```

(continues on next page)

(continued from previous page)

```
                    "FileInodeChangeDate": "...",
                    "FilePermissions": 600,
                    "FileType": "Win32 EXE",
                    "FileTypeExtension": "EXE",
                    "MIMEType": "application/octet-stream",
                    "MachineType": 332,
                    "TimeStamp": "2016:07:15 21:44:45-04:00",
                    "ImageFileCharacteristics": 258,
                    "PEType": 267,
                    "LinkerVersion": 14.0,
                    "CodeSize": 8192,
                    "InitializedDataSize": 22528,
                    "UninitializedDataSize": 0,
                    "EntryPoint": 10496,
                    "OSVersion": 10.0,
                    "ImageVersion": 10.0,
                    "SubsystemVersion": 10.0,
                    "Subsystem": 2,
                    "FileVersionNumber": "10.0.14393.0",
                    "ProductVersionNumber": "10.0.14393.0",
                    "FileFlagsMask": 63,
                    "FileFlags": 0,
                    "FileOS": 262148,
                    "ObjectFileType": 1,
                    "FileSubtype": 0,
                    "LanguageCode": "0409",
                    "CharacterSet": "04B0",
                    "CompanyName": "Microsoft Corporation",
                    "FileDescription": "Windows Calculator",
                    "FileVersion": "10.0.14393.0 (rs1_release.160715-1616)",
                    "InternalName": "CALC",
                    "LegalCopyright": "© Microsoft Corporation. All rights reserved.",
                    "OriginalFileName": "CALC.EXE",
                    "ProductName": "Microsoft® Windows® Operating System",
                    "ProductVersion": "10.0.14393.0",
                    "Warning": "Possibly corrupt Version resource"
                },
                "hash": {
                    "sha256":
↪"c74f41325775de4777000161a057342cc57a04e8b7be17b06576412eff574dc5",
                    "md5": "40e85286357723f326980a3b30f84e4f",
                    "sha1": "2e391131f9b77a8ec0e0172113692f9e2ccceaf0"
                }
            }
            "archivers": {},
            "plugins_run": {
                "workers": [
                    [
                        "exif",
                        "hash"
                    ]
                ],
                "archivers": []
            },
            "extracted_from": "e777051a-832b-489f-b74c-9949b2c9a2ce",
            "extracted_by": "pecarve"
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "payload_id": "5a6279a4-df1d-4575-8587-286f5938839d",
            "size": 507904,
            "payload_meta": {
                "should_archive": true,
                "extra_data": {
                    "offset": 50688
                },
                "dispatch_to": []
            },
            "workers": {
                "exif": {
                    "SourceFile": "/tmp/tmpsiaa54tm",
                    "ExifToolVersion": 11.11,
                    "FileName": "tmpsiaa54tm",
                    "Directory": "/tmp",
                    "FileSize": 507904,
                    "FileModifyDate": "...",
                    "FileAccessDate": "...",
                    "FileInodeChangeDate": "...",
                    "FilePermissions": 600,
                    "FileType": "Win32 EXE",
                    "FileTypeExtension": "EXE",
                    "MIMEType": "application/octet-stream",
                    "MachineType": 332,
                    "TimeStamp": "2013:04:20 10:50:10-04:00",
                    "ImageFileCharacteristics": 258,
                    "PEType": 267,
                    "LinkerVersion": 9.0,
                    "CodeSize": 386048,
                    "InitializedDataSize": 120832,
                    "UninitializedDataSize": 0,
                    "EntryPoint": 208320,
                    "OSVersion": 5.0,
                    "ImageVersion": 0.0,
                    "SubsystemVersion": 5.0,
                    "Subsystem": 2
                },
                "hash": {
                    "sha256":
↪"47c6e9b102324ea6c54dd95ad3fdf4b48b18775053b105e241a371a3731488c0",
                    "md5": "16d9f6e5491d99beb46d7ab1500c1799",
                    "sha1": "9e6414bf2802c98fbd13172817db80380c5eeb6a"
                }
            }
            "archivers": {},
            "plugins_run": {
                "workers": [
                    [
                        "exif",
                        "hash"
                    ]
                ],
                "archivers": []
            },
            "extracted_from": "e777051a-832b-489f-b74c-9949b2c9a2ce",
            "extracted_by": "pecarve"
```

(continues on next page)

(continued from previous page)

```
        }
    ],
    "request_meta": {
        "archive_payloads": true,
        "source": null,
        "extra_data": {}
    },
    "errors": {},
    "decorators": {},
    "scan_id": "04f9aec3-afc7-4fa1-b179-73e46c074e81"
}
```

1.6 Guides

1.6.1 User and Development Guide

Want to get started using *stoQ* or write your own plugins? Start reading here.

Installation

stoQ is extremely lightweight and strives for minimal dependencies. It can be installed either via *pip* or directly from source. Once you have stoQ installed, it's just a matter of installing the required plugins for your use case. stoQ has over 40 publicly available plugins that can be found in their own repository here.

Minimum requirements

stoQ requires a minimum of python 3.6 and is recommended to be run in a python venv.

Initial Setup

Setup a $STOQ_HOME (defaults to ~/.stoq) folder, the necessary plugin folder and a virtual environment:

```
$ mkdir -p ~/.stoq/plugins
$ python3 -m venv ~/.stoq/.venv
$ source ~/.stoq/.venv/bin/activate
```

Stable

The simplest way to get started is to install stoQ from pip:

```
$ pip3 install stoq-framework
```

Development

If you would rather use the latest development version, you can simply clone the repository and install from there:

```
$ git clone https://github.com/PUNCH-Cyber/stoq
```

Then, simply open the *stoq* directory and install:

```
$ cd stoq
$ python3 setup.py install
```

Note: Depending on your environment, you may also need to run *pip3 install wheel* to successfully install stoQ

Installing Plugins

There are two ways of installing *stoQ* plugins. All core public plugins can be installed via the command line directly from GitHub. Additionally, plugins can be installed from a local directory.

From GitHub

Once you have stoQ installed, you can start installing the publicly available plugins. For a full listing of plugins and a description of their functionality, you can visit the stoQ public plugins repository here.

In order to install plugins from the stoQ plugin repository, you can use the `stoq` command:

```
$ stoq install --github stoq:PLUGIN_NAME
```

For this example, let's just install the *yara* and *stdout* plugins. First, let's install the yara plugin:

```
$ stoq install --github stoq:yara
Successfully installed to ~/.stoq/plugins/yara
```

Now, let's install the stdout plugin:

```
$ stoq install --github stoq:stdout
Successfully installed to ~/.stoq/plugins/stdout
```

From directory

Plugins can also be installed from a local directory. This is useful if you have custom or third party plugins. Additionally, plugins can be installed from a cloned version of *stoQ's* public plugin repository:

```
$ stoq install path/to/plugin
```

Upgrading plugins

Plugins may be upgraded (or downgraded) by adding the *–upgrade* command line option to the install command:

```
$ stoq install --upgrade --github stoq:stdout
```

Warning: Upgrading plugins is a destructive operation. This will overwrite/remove all data within the plugins directory, to include the plugin configuration file. It is highly recommended that the plugin directory be backed up regularly to ensure important information is not lost, or plugin configuration options be defined in *stoq.cfg*.

Getting Started

Now that stoQ is installed, getting up and running is extremely simple. stoQ can be run a few different ways, depending on what your requirements are.

Configuring stoQ

stoq.cfg

stoQ's configuration file is not required, but does offer the convenience of overriding the default configuration. An example configuration file can be found here. By default, stoQ will look for `stoq.cfg` in `$STOQ_HOME` if running from the command line, or `$CWD` if being used as a library.

Plugin options may also be defined in `stoq.cfg`. More information on how to configure plugins in `stoq.cfg` can be found in *plugin configuration*.

$STOQ_HOME

When using the `stoq` command, stoQ will default to using `$HOME/.stoq` as it's home directory. This path is important as it is used as the default path for plugins and configuration files. You can easily override this by setting the `$STOQ_HOME` environment variable. For example, we can set stoQ's home directory to `/opt/stoq` like so:

```
$ export $STOQ_HOME=/opt/stoq
```

Now, stoQ will look for plugins in `/opt/stoq/plugins` and the `stoq.cfg` configuration file in `/opt/stoq/stoq.cfg`.

One thing to note is, `$STOQ_HOME` is only valid when using the `stoq` command. If you are using stoQ as a library, the default path will be `$CWD`.

Running stoQ

The easiest way to get started is by running stoQ from the command line. There are two modes available, *scan* and *run*. Before we get into what each more is used for, let's see how installed plugins can be listed.

List Plugins

Installed plugins can be easily listed by using the `stoq` command:

```
$ stoq list
stoQ :: v3.x.x :: an automated analysis framework
-------------------------------------------------
xdpcarve                      v3.0.0     Carve and decode streams from XDP documents
stdout                        v3.0.0     Sends content to STDOUT
rtf                           v3.0.0     Extract objects from RTF payloads
hash                          v3.0.0     Hash content
dirmon                        v3.0.0     Monitor a directory for newly created files␣
↪for processing
vtmis-search                  v3.0.0     Search VTMIS API
peinfo                        v3.0.0     Gather relevant information about an␣
↪executable using pefile
javaclass                     v3.0.0     Decodes and extracts information from Java␣
↪Class files
filedir                       v3.0.0     Ingest a file or directory for processing
yara                          v3.0.0     Process a payload using yara
decompress                    v3.0.0     Extract content from a multitude of archive␣
↪formats
ole                           v3.0.0     Carve OLE streams within Microsoft Office␣
↪Documents
iocextract                    v3.0.0     Regex routines to extract and normalize IOC
↪'s from a payload
mraptor                       v3.0.0     Port of mraptor3 from oletools
trid                          v3.0.0     Identify file types from their TrID signature
smtp                          v3.0.0     SMTP Parser Worker
exif                          v3.0.0     Processes a payload using ExifTool
pecarve                       v3.0.0     Carve portable executable files from a data␣
↪stream
swfcarve                      v3.0.0     Carve and decompress SWF files from a data␣
↪stream
```

Scan Mode

Scan mode is designed for scanning an individual payload from the command line. This is especially useful for lightweight tasks or one-off scans.

Let's get started. In this example, let's simply generate the MD5, SHA1, and SHA256 hashes of a file.

First, let's make sure we have the required plugins installed:

```
$ stoq install --github stoq:hash
```

Now, let's run `stoq` with the `hash` plugin:

```
$ stoq scan /tmp/bad.exe -s hash
{
```

(continues on next page)

(continued from previous page)

```
    "time": "...",
    "results": [
        {
            "payload_id": "0acfdfcf-f298-4950-96d2-13e3f93646b5",
            "size": 507904,
            "payload_meta": {
                "should_archive": true,
                "extra_data": {
                    "filename": "bad.exe"
                },
                "dispatch_to": []
            },
            "workers": {
                "hash": {
                    "sha256":
↪"47c6e9b402324ea6c54dd95ad3fdf4b48b18775053b105e241a371a3731488c0",
                    "md5": "16d9f6e5421d99beb46d7ab1500c1799",
                    "sha1": "9e6414bf28a2c98fbd13172817db80380c5eeb6a"
                }
            }
            "archivers": {},
            "plugins_run": {
                "workers": [
                    [
                        "hash"
                    ]
                ],
                "archivers": []
            },
            "extracted_from": null,
            "extracted_by": null
        }
    ],
    "request_meta": {
        "archive_payloads": true,
        "source": null,
        "extra_data": {}
    },
    "errors": {},
    "decorators": {},
    "scan_id": "5699d5ac-df3b-4ba1-bb38-296813d14d19"
}
```

Great, now we've generated the needed hashes; but stoQ allows us to do way more than just generate hashes. Let's also tell stoQ to use the `peinfo` plugin. First, let's make sure the plugin is installed:

```
$ stoq install --github stoq:peinfo
```

Ok, now let's scan the payload again, but this time we will use both plugins:

```
$ stoq scan /tmp/bad.exe -s hash peinfo
{
    "time": "...",
    "results": [
        {
            "payload_id": "38cb070d-c9e8-48be-84d9-6ee612489fe8",
```

(continues on next page)

(continued from previous page)

```
            "size": 507904,
            "payload_meta": {
                "should_archive": true,
                "extra_data": {
                    "filename": "bad.exe"
                },
                "dispatch_to": []
            },
            "workers": {
                "hash": {
                    "sha256":
↪"47c6e9b402324ea6c54dd95ad3fdf4b48b18775053b105e241a371a3731488c0",
                    "md5": "16d9f6e5421d99beb46d7ab1500c1799",
                    "sha1": "9e6414bf28a2c98fbd13172817db80380c5eeb6a"
                    }
                "peinfo": {
                    "imphash": "6238d5d3f08e2b63c437c2ba9e1f7151",
                    "compile_time": "2013-04-20 10:50:10",
                    "packer": null,
                    "is_packed": false,
                    "is_exe": true,
                    "is_dll": false,
                    "is_driver": false,
                    "is_valid": null,
                    "is_suspicious": null,
                    "machine_type": "IMAGE_FILE_MACHINE_I386",
                    "entrypoint": "0x32dc0",
                    "section_count": 5,
                    [...TRUNCATED...]
                }
            }
            "archivers": {},
            "plugins_run": {
                "workers": [
                    [
                        "hash",
                        "peinfo"
                    ]
                ],
                "archivers": []
            },
            "extracted_from": null,
            "extracted_by": null
        }
    ],
    "request_meta": {
        "archive_payloads": true,
        "source": null,
        "extra_data": {}
    },
    "errors": {},
    "decorators": {},
    "scan_id": "43f3210b-b4ce-41e5-b39a-5fb8dbbc45ac"
}
```

Now, you've run the payload with two different plugins simply by adding it to your command line. As you use stoQ, you will see the power this affords you. This is especially true when you start delving into some of the more advanced use cases. There are quite a few other command line options, we've only just scratched the surface. For more command line options available in *scan mode*, just run:

```
$ stoq scan -h
```

Run Mode

Run mode is similar to *scan mode*, but is meant for handling multiple payloads or for long running tasks. This mode requires the use of a *provider* plugin.

For this example, we will monitor a directory for new files. When new files are created, the plugin will detect this and send the payload to stoQ for scanning. Chances are we won't want the results to simply be displayed to the console, so we will also save the results to disk.

First, let's make sure the required plugins are installed. Let's start with the `dirmon` plugin. This plugin monitors a directory for newly created files:

```
$ stoq install --github stoq:dirmon
```

Now, time to install the `filedir` plugin. This plugin will save the results to disk:

```
$ stoq install --github stoq:filedir
```

We'll monitor the directory `/tmp/monitor` for this example and save our results to `/tmp/results`. Let's create these directories:

```
$ mkdir /tmp/monitor /tmp/results
```

Since we already have the `hash` and `peinfo` plugins installed from the *scan mode* example above, let's use them for scanning the payloads.:

```
$ stoq run -P dirmon -C filedir -a hash peinfo \
    --plugin-opts dirmon:source_dir=/tmp/monitor \
    filedir:results_dir=/tmp/results
```

Now, let's copy `bad.exe` into the monitor directory:

```
$ cp /tmp/bad.exe /tmp/monitor
```

Ok, stoQ should have detected `bad.exe` was created in `/tmp/monitor` and then scan the content with the `hash` and `peinfo` plugins, then save the results to `/tmp/results`. Let's take a look:

```
$ ls /tmp/results/
1f168f68-1c19-46f9-9427-585345a6fe24
```

Great! We have successfully monitored a directory for new files, scanned them with two plugins, and then saved the results to disk. Again, we've only scratched the surface as to what stoQ can do. For more command line options in *run* mode, simply run:

```
$ stoq run -h
```

Plugin configuration

Plugin configurations may be defined in several ways, see *plugin configuration*.

RequestMeta Options

RequestMeta options sets metadata associated with the initial request *stoQ* receives. This is useful when certain metadata, such as the source name of the payload, must be saved alongside the results of the scan.

There are two command line options avaiable for RequestMeta.

- `--request-source`
- `--request-extra`

To set `--request-source` simply add the argument to the `stoq` command:

```
$ stoq scan [...] --request-source my_mail
{
    "results": {
        {
            [...]
            "payload_id": "27774a9a-5a03-4d59-b51b-37583683b666",
            [...]
        }
    }
    "request_meta": {
        "archive_payloads": true,
        "source": "my_mail",
        "extra_data": {}
    },
    "errors": {},
    "time": "...",
    "decorators": {},
    "scan_id": "e107f362-0b40-455e-bfef-da7c606637ca"
}
```

Additionally, extra data may be added to RequestMeta by using the `--request-extra` command line argument. This option requires key/value pairs separated by an =:

```
$ stoq scan [...] --request-source my_mail --request-extra server=mail-server-01␣
↪postfix=true
{
    "results": {
        {
            [...]
            "payload_id": "27774a9a-5a03-4d59-b51b-37583683b666",
            [...]
        }
    }
    "request_meta": {
        "archive_payloads": true,
        "source": "my_mail",
        "extra_data": {
            "server": "mail-server-01",
            "postfix": true
        }
```

(continues on next page)

(continued from previous page)

```
    },
    "errors": {},
    "time": "...",
    "decorators": {},
    "scan_id": "e107f362-0b40-455e-bfef-da7c606637ca"
}
```

Additionally, RequestMeta may be defined when scanning a payload using a `Stoq` object:

```
>>> import asyncio
>>> from stoq import Stoq, RequestMeta
>>> s = Stoq()
>>> loop = asyncio.get_event_loop()
>>> request_meta = RequestMeta(source='my_mail', extra_data={'server': 'mail-server-01
↪', 'postfix': True})
>>> results = loop.run_until_complete(
...     s.scan(b'this is a test payload', request_meta=request_meta)
... )
```

Development

Core

Overview

stoQ is an extremely flexible framework. In this section we will go over some of the most advanced uses and show examples of how it can be used as a framework.

Framework

stoQ is much more than simply a command to be run. First and foremost, stoQ is a framework. The command *stoq* is simply a means of interacting with the framework. For more detailed and robust information on APIs available for stoQ, please check out the *plugin documentation*.

`Stoq` is the primary class for interacting with *stoQ* and its plugins. All arguments, except for plugins to be used, must be defined upon instantiation. Plugins can be loaded at any time. However, to ensure consistent behavior, it is recommended that all required plugins be loaded upon instantiation.

For these examples, it is assumed the below *plugins have been installed* in *$CWD/plugins*:

- dirmon
- exif
- filedir
- hash
- yara

Individual Scan

Individual scans are useful for scanning single payloads at a time. The user is responsible for ensuring a payload is passed to the `Stoq` class.

Note: `Provider` plugins are ignored when conducting an individual scan.

1. First, import the required class:

```
>>> import asyncio
>>> from stoq import Stoq, RequestMeta
```

2. We will now define the plugins we want to use. In this case, we will be loading the `hash`, and `exif` plugins:

```
>>> workers = ['hash', 'exif']
```

3. Now that we have our environment defined, lets instantiate the `Stoq` class:

```
>>> s = Stoq(always_dispatch=workers)
```

4. We can now load a payload, and scan it individually with *stoQ*:

```
>>> src = '/tmp/bad.exe'
>>> loop = asyncio.get_event_loop()
>>> with open(src, 'rb') as src_payload:
...     meta = RequestMeta(extra_data={'filename': src})
...     results = loop.run_until_complete(s.scan(
...             content=src_payload.read(),
...             request_meta=meta))
>>> print(results)
...    {
...        "time": "...",
...        "results": [
...            {
...                "payload_id": "...",
...                "size": 507904,
...                "payload_meta": {
...                    "should_archive": true,
...                    "extra_data": {
...                        "filename": "/tmp/bad.exe"
...                    },
...                    "dispatch_to": []
...                },
...                "workers": {
...                        "hash": {
... [...]
```

Using Providers

Using stoQ with providers allows for the scanning of multiple payloads from multiple sources. This method will instantiate a *Queue* which payloads or requests are published to for scanning by *stoQ*. Additionally, payloads may be retrieved from multiple disparate data sources using *Archiver* plugins.

1. First, import the required class:

```
>>> import asyncio
>>> from stoq import Stoq
```

2. We will now define the plugins we want to use. In this case, we will be loading the `dirmon`, `filedir`, `hash`, and `exif` plugins. We will also set the `base_dir` to a specific directory. Additionally, we will also set some plugin options to ensure the plugins are operating the way we'd like them:

```
>>> always_dispatch = ['hash']
>>> providers = ['dirmon']
>>> connectors = ['filedir']
>>> dispatchers = ['yara']
>>> plugin_opts = {
...     'dirmon': {'source_dir': '/tmp/datadump'},
...     'filedir': {'results_dir': '/tmp/stoq-results'}
... }
>>> base_dir = '/usr/local/stoq'
>>> plugin_dirs = ['/opt/plugins']
```

Note: Any plugin options available in the plugin's `.stoq` configuration file can be set via the `plugin_opts` argument.

3. Now that we have our environment defined, lets instantiate the `Stoq` class, and run:

```
>>> s = Stoq(
...     base_dir=base_dir,
...     plugin_dir_list=plugin_dirs,
...     dispatchers=dispatchers,
...     providers=providers,
...     connectors=connectors,
...     plugins_opts=plugins_opts,
...     always_dispatch=always_dispatch
... )
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(s.run())
```

A few things are happening here:

1. The `/tmp/datadump` directory is being monitored for newly created files
2. Each file is opened, and the payload is loaded into `Stoq` asynchronously
3. The payload is scanned with the `yara` dispatcher plugin
4. The yara dispatcher plugin returns a list of plugins that the payload should be scanned with
5. The plugins identified by the `yara` dispatcher are loaded, and the payload is sent to them
6. Each payload will always be sent to the `hash` plugin because it was defined in `always_dispatch`
7. The results from all plugins are collected, and sent to the `filedir` connector plugin

8. The `filedir` plugin saves each result to disk in `/tmp/stoq-results`

Manual Interaction

`Stoq` may also be interacted with manually, rather than relying on the normal workflow. In this section, we will touch on how this can be done.

Instantiating stoQ

Let's start by simply instantiating `Stoq` with no options. There are several arguments available when instantiating `Stoq`, please refer to the *plugin documentation* for more information and options available.:

```
>>> from stoq import Stoq
>>> s = Stoq()
```

Loading plugins

stoQ plugins can be loaded using a simple helper function. The framework will automatically detect the type of plugin is it based on the `class` of the plugin. There is no need to define the plugin type, *stoQ* will handle that once it is loaded.:

```
>>> plugin = s.load_plugin('yara')
```

Instantiate Payload Object

In order to scan a payload, a `Payload` object must first be instantiated. The `Payload` object houses all information related to a payload, to include the content of the payload and metadata (i.e., size, originating plugin information, dispatch metadata, among others) pertaining to the payload. Optionally, a `Payload` object can be instantiated with a `PayloadMeta` object to ensure the originating metadata (i.e., filename, source path, etc...) is also made available:

```
>>> import os
>>> import asyncio
>>> from stoq.data_classes import PayloadMeta, Payload
>>> filename = '/tmp/test_file.exe'
>>> with open(filename, 'rb') as src:
...     meta = PayloadMeta(
...         extra_data={
...             'filename': os.path.basename(filename),
...             'source_dir': os.path.dirname(filename),
...         }
...     )
>>> payload = Payload(src.read(), meta)
```

Scan payload

There are two helper functions available for scanning a payload. If a dispatcher plugin is not being used, then a worker plugin must be defined by passing the `add_start_dispatch` argument. This tells *stoQ* to send the `Payload` object to the specified worker plugins.

From raw bytes

If a *Payload* object has not been created yet, the content of the raw payload can simply be passed to the *Stoq.scan* function. A `Payload` object will automatically be created.:

```
>>> loop = asyncio.get_event_loop()
>>> start_dispatch = ['yara']
>>> results = loop.run_until_complete(
...     s.scan('raw bytes', add_start_dispatch=start_dispatch)
... )
```

From `Payload` object

If a `Payload` object has already been instantiated, as detailed above, the `scan_request` function may be called. First, a new *Request* object must be instantiated with the *Payload* object that we previously created:

```
>>> import asyncio
>>> from stoq import Payload, Request, RequestMeta
>>> start_dispatch = ['yara']
>>> loop = asyncio.get_event_loop()
>>> payload = Payload(b'content to scan')
>>> request = Request(payloads=[payload], request_meta=RequestMeta())
>>> results = loop.run_until_complete(
...    s.scan_request(request, add_start_dispatch=start_dispatch)
... )
```

Save Results

Finally, results may be saved using the desired `Connector` plugin. *stoQ* stores results from the framework as a `StoqResponse` object. The results will be saved to all connector plugins that have been loaded. In this example, we will only load the `filedir` plugin which will save the results to a specified directory.:

```
>>> connector = s.load_plugin('filedir')
>>> loop.run_until_complete(connector.save(results))
```

Split Results

In some cases it may be required to split results out individually. For example, when saving results to different indexes depending on plugin name, such as with ElasticSearch or Splunk.

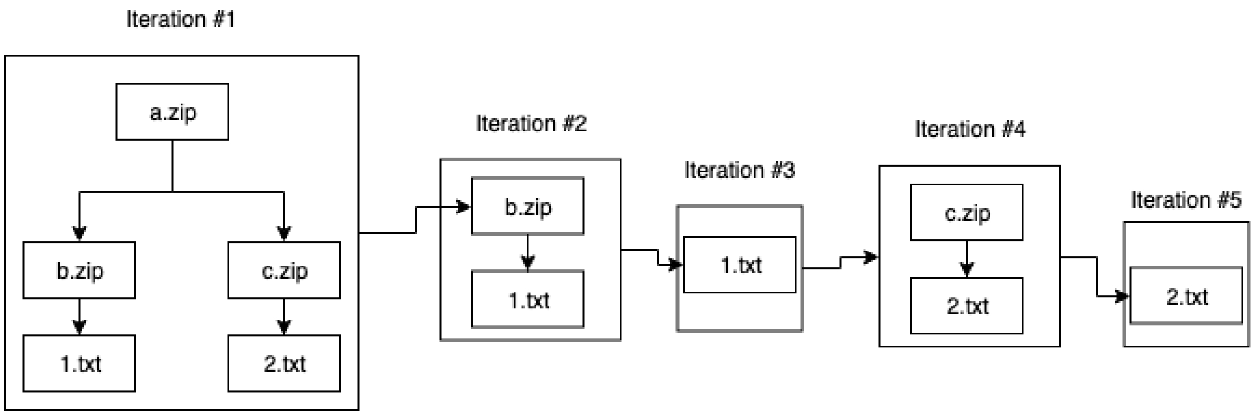
```
>>> results = loop.run_until_complete(s.scan(payload))
>>> split_results = results.split()
```

Reconstructing Subresponse Results

stoQ can produce complex results depending on the recursion depth and extracted payload objects. In order to help handle complex results and limit redundant processing of payloads when using stoQ as a framework, a method exists that will allow for iterating over each result as if it were the original root object. This is especially useful when handling compressed archives, such as *zip* or *apk* files that may have multiple levels of archived content. Additionally, the defined decorators will be run against each newly constructed *StoqResponse* and added to the results.

```
>>> await for result in s.reconstruct_all_subresponses(results):
...     print(result)
```

Below is a simple flow diagram of the iterated results when being reconstructed.



Multiple Plugin directories

When instantiating `Stoq`, multiple plugins directories may be defined. For more information on default paths, please refer to the *getting started documentation*:

```
>>> from stoq import Stoq
>>> plugin_directories = ['/usr/local/stoq/plugins', '/home/.stoq/plugins']
>>> s = Stoq(plugin_dir_list=plugin_directories)
```

API

class stoq.core.**Stoq**(*base_dir=None*, *config_file=None*, *log_dir=<object object>*, *log_level=None*, *plugin_dir_list=None*, *plugin_opts=None*, *providers=None*, *provider_consumers=None*, *source_archivers=None*, *dest_archivers=None*, *connectors=None*, *dispatchers=None*, *decorators=None*, *always_dispatch=None*, *max_queue=None*, *max_recursion=None*, *max_required_worker_depth=None*)

Core Stoq Class

Parameters

- **base_dir** (`Optional[str]`) – Base directory for stoQ
- **config_file** (`Optional[str]`) – stoQ Configuration file
- **log_dir** (`object`) – Path to log directory
- **log_level** (`Optional[str]`) – Log level for logging events
- **plugin_dir_list** (`Optional[List[str]]`) – Paths to search for stoQ plugins
- **plugin_opts** (`Optional[Dict[str, Dict]]`) – Plugin specific options that are passed once a plugin is loaded
- **providers** (`Optional[List[str]]`) – Provider plugins to be loaded and run for sending payloads to scan
- **source_archivers** (`Optional[List[str]]`) – Archiver plugins to be used for loading payloads for analysis
- **dest_archiver** – Archiver plugins to be used for archiving payloads and extracted payloads
- **connectors** (`Optional[List[str]]`) – Connectors to be loaded and run for saving results
- **dispatchers** (`Optional[List[str]]`) – Dispatcher plugins to be used
- **decorators** (`Optional[List[str]]`) – Decorators to be used
- **always_dispatch** (`Optional[List[str]]`) – Plugins to always send payloads to, no matter what
- **provider_consumers** (`Optional[int]`) – Number of provider consumers to instaniate
- **max_queue** (`Optional[int]`) – Max Queue size for Providers plugins
- **max_recursion** (`Optional[int]`) – Maximum level of recursion into a payload and extracted payloads
- **max_required_worker_depth** (`Optional[int]`) – Maximum depth for required worker plugins dependencies

reconstruct_all_subresponses(*stoq_response*)

Generate a new *StoqResponse* object for each *Payload* within the *Request*

Return type `AsyncGenerator[StoqResponse, None]`

async run(*request_meta=None*, *add_start_dispatch=None*)

Run stoQ using a provider plugin to scan multiple files until exhaustion

Parameters

- **request_meta** (`Optional[RequestMeta]`) – Metadata pertaining to the originating request
- **add_start_dispatch** (`Optional[List[str]]`) – Force first round of scanning to use specified plugins

Return type `None`

async scan(*content*, *payload_meta=None*, *request_meta=None*, *add_start_dispatch=None*, *ratelimit=None*)

Wrapper for *scan_request* that creates a *Payload* object from bytes

Parameters

- **content** (`bytes`) – Raw bytes to be scanned
- **payload_meta** (`Optional[PayloadMeta]`) – Metadata pertaining to originating source
- **request_meta** (`Optional[RequestMeta]`) – Metadata pertaining to the originating request
- **add_start_dispatch** (`Optional[List[str]]`) – Force first round of scanning to use specified plugins
- **ratelimit** (`Optional[str]`) – Rate limit calls to scan

Return type `StoqResponse`

async scan_request(*request*, *add_start_dispatch=None*)

Scan an individual payload

Parameters

- **request** (`Request`) – `Request` object of payload(s) to be scanned
- **add_start_dispatch** (`Optional[List[str]]`) – Force first round of scanning to use specified plugins

Return type `StoqResponse`

Exceptions

exception `stoq.exceptions.`**StoqException**

exception `stoq.exceptions.`**StoqPluginNotFound**

exception `stoq.exceptions.`**StoqPluginException**

Plugins

Overview

stoQ is a highly flexible framework because of its ability to leverage plugins for each layer of operations. One of the biggest benefits to this approach is that it ensures the user is able to quickly and easily pivot to and from different technologies in their stack, without having to drastically alter workflow.

For a full listing of all publicly available plugins, check out the stoQ public plugins repository.

Configuration

Plugins may be provided configuration options in one of four ways. In order of precendece:

- From the command line
- Upon instantiation of `Stoq()`
- Defined in `stoq.cfg`
- Defined in the plugin's `.stoq` configuration file

Command Line

When running `stoq` from the command line, simply add `--plugin-opts` to your arguments followed by the desired plugin options. The syntax for plugin options is:

```
plugin_name:option=value
```

For example, if we want to tell the plugin `dirmon` to monitor the directory `/tmp/monitor` for new files by setting the option `source_dir`, the syntax would be:

```
dirmon:source_dir=/tmp/monitor
```

Instantiation

When using stoQ as a framework, plugin options may be defined when instantiating `Stoq` using the `plugin_opts` argument:

```
>>> from stoq import Stoq
>>> plugin_options = {'dirmon': {'source_dir': '/tmp/monitor'}}
>>> s = Stoq(plugin_opts=plugin_options)
```

stoq.cfg

The recommended location for storing static plugin configuration options is in `stoq.cfg`. The reason for this if all plugin options defined in the plugin's `.stoq` file will be overwritten when the plugin is upgraded.

To define plugin options in `stoq.cfg` simply add a section header of the plugin name, then define the plugin options. For example, to define the plugin option `source_dir` for the `dirmon` plugin, the below can be added to `stoq.cfg`:

```
[dirmon]
source_dir = /tmp/monitor
```

Plugin .stoq configuration file

Each plugin must have a `.stoq` configuration file. The configuration file resides in the same directory as the plugin module. The plugin's configuration file allows for configuring a plugin with default or static settings. The configuration file is a standard YAML file and is parsed using the `configparser` module. The following is an example plugin configuration file with all *required* fields:

```
[Core]
Name = example_plugin
Module = example_plugin

[Documentation]
Author = PUNCH Cyber
Version = 0.1
Website = https://github.com/PUNCH-Cyber/stoq-plugins-public
Description = Example stoQ Plugin
```

- **Core**
  - **Name**: The plugin name that stoQ will use when calling the plugin. This must be unique.
  - **Module**: The python module that contains the plugin (without the *.py* extension).
- **Documentation**
  - **Author**: Author of the plugin
  - **Version**: Plugin version
  - **Website**: Website where the plugin can be found
  - **Description**: Description of the plugins utility

Additionally, some optional settings may be defined:

```
[options]
min_stoq_version = 3.0.0
```

- **options**
  - **min_stoq_version**: Minimum version of stoQ required to work properly. If the version of *stoQ* is less than the version defined, a warning will be raised.

Note: Plugin options *must* be under the *[options]* section header to be accessible via the other plugin configuration options.

Warning: Plugin configuration options may be overwritten when a plugin is upgraded. Upgrading plugins is a destructive operation. This will overwrite/remove all data within the plugins directory, to include the plugin configuration file. It is highly recommended that the plugin directory be backed up regularly to ensure important information is not lost, or plugin configuration options be defined in *stoq.cfg*.

Multiclass Plugins

Plugins that are of more than one plugin class are called *Multiclass Plugins*. *Multiclass plugins* help to simplify and centralize plugin code. Development is nearly identical to creating a regular plugin.In order to create a *Multiclass plugin*, the plugin must be a subclass of one or more plugin class.

In this example, we will create a *Multiclass plugin* that is a worker as well as a dispatcher plugin. We simply need to subclass our plugin class with `WorkerPlugin` and `DispatcherPlugin` and ensure the `scan` (required for worker plugins) and `get_dispatches` (required for dispatcher plugins) methods exist.:

```
from typing import Optional
from stoq import Payload, Request, WorkerResponse
from stoq.plugins import DispatcherPlugin, WorkerPlugin

class MultiClassPlugin(WorkerPlugin, DispatcherPlugin):
    async def scan(
        self, payload: Payload, request: Request
    ) -> Optional[WorkerResponse]:
        # do worker plugin stuff here
        return

    async def get_dispatches(
        self, payload: Payload, request: Request
    ) -> Optional[DispatcherResponse]:
        # do dispatcher plugin stuff here
        return
```

Plugin Logging

Upon instantiation, plugins are provided a `Logger` object within the plugin class named `self.log`. This is just a standard Python logging object that supports the log levels `debug`, `info`, `warning`, `error`, and `critical`.:

```
from typing import Optional
from stoq.plugins import WorkerPlugin
from stoq import Payload, Request, WorkerResponse

class LoggingPlugin(WorkerPlugin):
    async def scan(
        self, payload: Payload, request: Request
    ) -> Optional[WorkerResponse]:
        self.log.info('Scanning payload now')
```

Errors

Errors from plugins must be handled with the `Error` class. This helps to ensure a consistent and standardized error message handling across the framework. All plugin classes are capable of handling errors, except for the `ConnectorPlugin` class. The following is an example of adding a error to a `WorkerResponse`.:

```
from typing import Optional
from stoq.plugins import WorkerPlugin
from stoq import Error, Payload, Request, WorkerResponse

class ErrorPlugin(WorkerPlugin):
```

(continues on next page)

(continued from previous page)

```
    async def scan(
        self, payload: Payload, request: Request
    ) -> Optional[WorkerResponse]:
        errors: List[Error] = []
        errors.append(
            Error(
                error='This is an error message that will be in StoqResponse',
                plugin_name=self.plugin_name,
                payload_id=payload.results.payload_id
            )
        )
        return WorkerResponse(errors=errors)
```

Classes

Archiver Plugins

Overview

Archiver plugins are used for retrieving or saving scanned payloads. A payload can be anything from the initial payload scanned, or extracted payloads from previous scans. There are two types of archivers, *source* and *destination*.

destination

Archiver plugins used as a destination useful for saving payloads, be it the original scanned payload or any extracted payloads. Multiple destination archivers can be defined, allowing for a payload to be saved in either a single or multiple locations. The results from this plugin method may be used to subsequently load the payload again.

Destination archiver plugins can be defined multiple ways. In these examples, we will use the `filedir` archiver plugin.

From `stoq.cfg`:

```
[core]
dest_archivers = filedir
```

Note: Multiple plugins can be defined separated by a comma

From the command line:

```
$ stoq run -A filedir [...]
```

Note: Multiple plugins can be defined by simply adding the plugin name

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> dest_archivers = ['filedir']
>>> s = Stoq(dest_archivers=dest_archivers)
```

source

Archiver plugins used as a source retrieve payloads for scanning. This is useful in several use cases, such as when using a provider plugin that isn't able to pass a payload to *stoQ*. For example, if the provider plugin being used leverages a queueing system, such as RabbitMQ, there may be problems placing multiple payloads onto a queue as it is inefficient, prone to failure, and does not scale well. With archiver plugins as a source, the queuing system can be leveraged by sending a message with a payload location, and the archiver plugin can then retrieve the payload for scanning. The *ArchiverResponse* results returned from *ArchiverPlugin.archive()* is used to load the payload.

Source archiver plugins can be defined multiple ways. In these examples, we will use the `filedir` archiver plugin.

From `stoq.cfg`:

```
[core]
source_archivers = filedir
```

Note: Multiple plugins can be defined separated by a comma

From the command line:

```
$ stoq run -S filedir [...]
```

Note: Multiple plugins can be defined by simply adding the plugin name

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> source_archivers = ['filedir']
>>> s = Stoq(source_archivers=source_archivers)
```

Writing a plugin

Unlike most other *stoQ* plugins, *archiver* plugins have two core methods, of which at least one of the below is required.

- archive
- get

The `archive` method is used to archive payloads that are passed to *stoQ* or extracted from other plugins. In order for a payload to be archived, that attribute `should_archive` must be set to `True` in the payloads `PayloadMeta` object. If set to `False`, the payload will not be archived.

An *archiver* plugin must be a subclass of the `ArchiverPlugin` class.

As with any plugin, a *configuration file* must also exist and be properly configured.

Example

```
from typing import Dict, Optional

from stoq.plugins import ArchiverPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import ArchiverResponse, Payload, Request, PayloadMeta


class ExampleArchiver(ArchiverPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
        self.archive_path = config.get(
            'options', 'archive_path', fallback='/tmp/archive_payload')

    async def archive(
        self, payload: Payload, request: Request
    ) -> Optional[ArchiverResponse]:
        with open(f'{self.archive_path}', 'wb) as out:
            out.write(payload.content)
        ar = ArchiverResponse({'path': f'{self.archive_path}'})
        return ar

    async def get(self, task: ArchiverResponse) -> Optional[Payload]:
        with open(task.results['path'], 'rb') as infile:
            return Payload(
                infile.read(),
                PayloadMeta(
                    extra_data={'path': task.results['path']}))
```

Note: *ArchiverPlugin.archive()* returns an *ArchiverResponse* object, which contains metadata that is later used by *ArchiverPlugin.get()* to load the payload.

API

class stoq.plugins.archiver.**ArchiverPlugin**(*config*)

async archive(*payload, request*)
Archive payload

Parameters

- **payload** (Payload) – Payload object to archive
- **request** (Request) – Originating Request object

Return type Optional[*ArchiverResponse*]

Returns ArchiverResponse object. Results are used to retrieve payload.

```
>>> import asyncio
>>> from stoq import Stoq, Payload
>>> payload = Payload(b'this is going to be saved')
>>> s = Stoq()
```

(continues on next page)

(continued from previous page)

```
>>> loop = asyncio.get_event_loop()
>>> archiver = s.load_plugin('filedir')
>>> loop.run_until_complete(archiver.archive(payload))
... {'path': '/tmp/bad.exe'}
```

async get(*task*)
Retrieve payload for processing

Parameters **task** (*ArchiverResponse*) – Task to be processed to load payload. Must contain *ArchiverResponse* results from *ArchiverPlugin.archive()*

Return type `Optional[Payload]`

Returns Payload object for scanning

```
>>> import asyncio
>>> from stoq import Stoq, ArchiverResponse
>>> s = Stoq()
>>> loop = asyncio.get_event_loop()
>>> archiver = s.load_plugin('filedir')
>>> task = ArchiverResponse(results={'path': '/tmp/bad.exe'})
>>> payload = loop.run_until_complete(archiver.get(task))
```

Response

class stoq.data_classes.**ArchiverResponse**(*results=None*, *errors=None*)
Object containing response from archiver destination plugins

Parameters

- **results** (`Optional[Dict]`) – Results from archiver plugin
- **errors** (`Optional[List[Error]]`) – Errors that occurred

```
>>> from stoq import ArchiverResponse
>>> results = {'file_id': '12345'}
>>> archiver_response = ArchiverResponse(results=results)
```

Connector Plugins

Overview

The last plugin class is the Connector plugin. This plugin class allows for the saving or passing off of the final result. Once all other plugins have completed their tasks, the final result is sent to the loaded connector plugins for handling. For example, a connector plugin may save results to disk, ElasticSearch, or even pass them off to a queueing system such as RabbitMQ.

Connector plugins can be defined multiple ways. In these examples, we will use the `filedir` connector plugin, allowing results to be saved to disk.

From `stoq.cfg`:

```
[core]
connectors = filedir
```

Note: Multiple plugins can be defined separated by a comma.

From the command line:

```
$ stoq run -C filedir [...]
```

Note: Multiple plugins can be defined by simply adding the plugin name

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> connectors = ['filedir']
>>> s = Stoq(connectors=connectors, [...])
```

Writing a plugin

A *connector* plugin must be a subclass of the `ConnectorPlugin` class.

As with any plugin, a *configuration file* must also exist and be properly configured.

Example

```
from typing import Dict, Optional

from stoq.plugins import ConnectorPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import StoqResponse

class ExampleConnector(ConnectorPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
        self.output_file = config.get(
            'options', 'output_file', fallback='/tmp/stoqresult.txt')

    async def save(self, response: StoqResponse) -> None:
        with open(f'{self.output_file}', 'w') as result:
            result.write(response)
```

API

class `stoq.plugins.connector.`**`ConnectorPlugin`**(*config*)

abstract async save(*response*)

Return type `None`

Decorator Plugins

Overview

Decorator plugins are the last plugins run just before saving results. This plugin class allows for the analysis of all results from each plugin, the original payload, and any extracted payloads. Multiple decorator plugins can be loaded, but each plugin is only passed the results once. Decorator plugins are extremely useful when post-processing is required of the collective results from the entire stoQ workflow.

Decorator plugins can be defined multiple ways. In these examples, we will use the `test_decorator` decorator plugin.

From `stoq.cfg`:

```
[core]
decorators = test_decorator
```

Note: Multiple plugins can be defined separated by a comma.

From the command line:

```
$ stoq run -D yara [...]
```

Note: Multiple plugins can be defined by simply adding the plugin name

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> decorators = ['test_decorator']
>>> s = Stoq(decorators=decorators, [...])
```

Writing a plugin

A *decorator* plugin must be a subclass of the `DecoratorPlugin` class. Results from a decorator are appended to the final `StoqResponse` object.

As with any plugin, a *configuration file* must also exist and be properly configured.

Example

```
from typing import Dict, Optional

from stoq.plugins import DecoratorPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import StoqResponse, DecoratorResponse


class ExampleDecorator(DecoratorPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
```

(continues on next page)

(continued from previous page)

```
        self.msg = config.get('options', 'msg', fallback='do_more msg')

    async def decorate(self, response: StoqResponse) -> Optional[DecoratorResponse]:
        do_more = False
        if 'yara' in response.results[0].plugins_run:
            do_more = True
        dr = DecoratorResponse({'do_more': do_more, 'msg': self.msg})
        return dr
```

API

class `stoq.plugins.decorator.`**`DecoratorPlugin`**(*config*)

  abstract async decorate(*response*)

    Return type `Optional[`*`DecoratorResponse`*`]`

Response

class `stoq.data_classes.`**`DecoratorResponse`**(*results=None*, *errors=None*)

  Object containing response from decorator plugins

  Parameters

  - **results** (`Optional[Dict]`) – Results from decorator plugin
  - **errors** (`Optional[List[Error]]`) – Errors that occurred

```
>>> from stoq import DecoratorResponse
>>> results = {'decorator_key': 'decorator_value'}
>>> errors = ['This plugin failed for a reason']
>>> response = DecoratorResponse(results=results, errors=errors)
```

Dispatcher Plugins

Overview

Dispatcher plugins allow for dynamic routing and loading of worker plugins. These plugins are extremely powerful in that they allow for an extremely flexible scanning flow based on characteristics of the payload itself. For instance, routing a payload to a worker plugin for scanning can be done by yara signatures, TRiD results, simple regex matching, or just about anything else. Each loaded dispatcher plugin is run once per payload.

Dispatcher plugins can be defined multiple ways. In these examples, we will use the `yara` dispatcher plugin.

From `stoq.cfg`:

```
[core]
dispatchers = yara
```

Note: Multiple plugins can be defined separated by a comma

From the command line:

```
$ stoq run -R yara [...]
```

Note: Multiple plugins can be defined by simply adding the plugin name

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> dispatchers = ['yara']
>>> s = Stoq(dispatchers=dispatchers, [...])
```

Now, let's write a simple yara rule to pass a payload to the `pecarve` plugin if a DOS stub is found:

```
rule exe_file
{
    meta:
        plugin = "pecarve"
        save = "True"
    strings:
        $MZ = "MZ"
        $ZM = "ZM"
        $dos_stub = "This program cannot be run in DOS mode"
        $win32_stub = "This program must be run under Win32"
    condition:
        ($MZ or $ZM) and ($dos_stub or $win32_stub)
}
```

In this case, if this yara signature hits on a payload, the payload will be passed to the `pecarve` plugin, which will then extract the PE file as a payload, and send it to *stoQ* for continued scanning. Additionally, because `save = "True"`, the extracted payload will also be saved if a *Destination Archiver* plugin is defined.

Writing a plugin

A *dispatcher* plugin must be a subclass of the `DispatcherPlugin` class.

As with any plugin, a *configuration file* must also exist and be properly configured.

Example

```
from typing import Dict, Optional

from stoq.plugins import DispatcherPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import Payload, DispatcherResponse, Request


class ExampleDispatcher(DispatcherPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
```

(continues on next page)

(continued from previous page)

```
        super().__init__(config)
        self.msg = config.get('options', 'msg', fallback='Useful content here')

    async def get_dispatches(
        self, payload: Payload, request: Request
    ) -> Optional[DispatcherResponse]:
        dr = DispatcherResponse()
        dr.meta['example_key'] = 'Useful metadata info'
        dr.meta['msg'] = self.msg
        return dr
```

API

class stoq.plugins.dispatcher.**DispatcherPlugin**(*config*)

abstract async get_dispatches(*payload*, *request*)

Return type Optional[*DispatcherResponse*]

Response

class stoq.data_classes.**DispatcherResponse**(*plugin_names=None*, *meta=None*, *errors=None*)

Object containing response from dispatcher plugins

Parameters

- **plugins_names** – Plugins to send payload to for scanning
- **meta** (Optional[Dict]) – Metadata pertaining to dispatching results
- **errors** (Optional[List[Error]]) – Errors that occurred

```
>>> from stoq import DispatcherResponse
>>> plugins = ['yara', 'exif']
>>> meta = {'hit': 'exe_file'}
>>> dispatcher = DispatcherResponse(plugin_names=plugins, meta=meta)
```

Provider Plugins

Overview

Provider plugins are designed for passing multiple payloads, or locations of payloads, to *stoQ*. They allow for multiple payloads to be run against *stoQ* until the source is exhausted. As such, they are useful for monitoring directories for new files, subscribing to a queue (i.e., RabbitMQ, Google PubSub, ZeroMQ), or scanning entire directories recursively. Multiple provider plugins can be provided allowing for even more flexibility. Provider plugins may either send a payload to *stoQ* for scanning, or send a message that an *Archiver plugin* is able to handle for loading of a payload.

Note: Provider plugins are not available when using *scan mode*. This is due to *scan mode* being designed for individual scans, not multiple payloads.

Provider plugins can be defined multiple ways. In these examples, we will use the `dirmon` provider plugin.

From `stoq.cfg`:

```
[core]
providers = dirmon
```

Note: Multiple plugins can be defined separated by a comma

From the command line:

```
$ stoq run -P dirmon [...]
```

Note: Multiple plugins can be defined by simply adding the plugin name

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> providers = ['dirmon']
>>> s = Stoq(providers=providers, [...])
```

Writing a plugin

Provider plugins add `Payload` or `Request` objects to the *stoQ* queue, or a `str`. If a `Payload` object is added, *stoQ* will begin processing the payload. If a `Request` object is added, *stoQ* will begin processing the request (which should contain at least one payload). If a `str` is added, *stoQ* will pass it to `Archiver` plugins that were loaded when `Stoq` was instantiated with the `source_archivers` argument.

A *provider* plugin must be a subclass of the `ProviderPlugin` class.

As with any plugin, a *configuration file* must also exist and be properly configured.

If a `Request` object is added to the queue and has *request_meta* set, then the *request_meta* passed to the `Stoq` *run()* method is ignored for this request.

Example

```
from asyncio import Queue
from typing import Dict, Optional

from stoq import Payload, PayloadMeta
from stoq.plugins import ProviderPlugin
from stoq.helpers import StoqConfigParser


class ExampleProvider(ProviderPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
        self.meta = config.get('options', 'meta', fallback='This msg will always be')

    async def ingest(self, queue: Queue) -> None:
```

(continues on next page)

(continued from previous page)

```
        payload_meta = PayloadMeta(extra_data={'msg': self.meta})
        await queue.put(Payload(b'This is a payload', payload_meta=payload_meta))
```

API

class stoq.plugins.provider.**ProviderPlugin**(*config*)

abstract async ingest(*queue*)

Return type None

Worker Plugins

Overview

Worker plugins are the primary data producers within *stoQ*. These plugins allow for tasks such as scanning payloads with yara, hashing payloads, and even extracting indicators of compromise (IOC) from documents. Worker plugins can be defined in all scanning modes. Additionally worker plugins can be dynamically loaded using dispatching plugins. More information on dispatcher plugins can be found in the *dispatcher plugin section*.

Worker plugins can be defined multiple ways. In these examples, we will use the `hash` worker plugin.

From the command line, worker plugins can be defined two different ways, depending on the use.

If *only* the original payload must be scanned, then `--start-dispatch` or `-s` command line argument may be used.:

```
$ stoq scan -s hash [...]
```

However, if the original payload and all subsequent payloads must be scanned, the `--always-dispatch` or `-a` command line argument may be used:

```
$ stoq scan -a hash [...]
```

Note: The difference between `--start-dispatch` and `--always-dispatch` can be somewhat confusing. The primary difference between the two is that if a worker plugin extracts any payloads for further scanning, any extracted payloads will only be scanned by workers defined by `--always-dispatch`. If `--start-dispatch` was used, the plugin defined will not be used to scan any extracted payloads.

Or, when instantiating the `Stoq()` class:

```
>>> import stoq
>>> workers = ['yara']
>>> s = Stoq(always_dispatch=workers, [...])
```

Lastly, worker plugins can be defined by dispatcher plugins. As mentioned previously, more information on them can be found in the *dispatcher plugin section*

Writing a plugin

A *worker* plugin must be a subclass of the `WorkerPlugin` class.

As with any plugin, a *configuration file* must also exist and be properly configured.

Example

```
from typing import Dict, List, Optional

from stoq.plugins import WorkerPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import (
    Payload,
    Request,
    WorkerResponse,
)


class ExampleWorker(WorkerPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
        self.useful = config.getboolean('options', 'useful', fallback=False)

    async def scan(
        self, payload: Payload, request: Request
    ) -> Optional[WorkerResponse]:
        response = {'worker_results': f'useful: {self.useful}'}
        return WorkerResponse(response)
```

Required Workers

required_workers is a configuration option specific to *WorkerPlugin* class. The purpose of this option is to allow a user to define worker dependencies. For example, WorkerA must be run after WorkerB because WorkerA requires the results from WorkerB to run successfully. This configuration option may be set in the *.stoq* configuration file for the *WorkerPlugin*, or within the *__init__* function.

```
from typing import List, Optional

from stoq.plugins import WorkerPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import (
    Payload,
    Request,
    WorkerResponse,
)
class WorkerA(WorkerPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
        self.required_workers = config.getset(
            'options', 'required_workers', fallback=set('WorkerB')
        )

    async def scan(
```

(continues on next page)

(continued from previous page)

```
        self, payload: Payload, request: Request
    ) -> Optional[WorkerResponse]:
        is_bad: bool = payload.results.workers['WorkerB']['is_bad']
        response = {'worker_results': f'is_bad: {is_bad}'}
        return WorkerResponse(response)
```

Extracted Payloads

Worker plugins may also extract payloads, and return them to `Stoq` for further analysis. Each extracted payload that is returned will be inserted into the same workflow as the original payload.

```
from typing import Dict, List, Optional

from stoq.plugins import WorkerPlugin
from stoq.helpers import StoqConfigParser
from stoq.data_classes import (
    ExtractedPayload,
    Payload,
    PayloadMeta,
    RequestMeta,
    WorkerResponse,
)


class ExampleWorker(WorkerPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)
        self.useful = config.getboolean('options', 'useful', fallback=False)

    async def scan(
        self, payload: Payload, request: Request
    ) -> Optional[WorkerResponse]:
        extracted_payloads: List = []
        extracted_payloads.append(ExtractedPayload(b'Lorem ipsum'))
        response = {'worker_results': f'useful: {self.useful}'}
        return WorkerResponse(response, extracted=extracted_payloads)
```

Dispatch To

In some cases it may be useful for a worker plugin to dicate which plugins an extracted payload is scanned with.

```
>>> meta = PayloadMeta(dispatch_to=['yara'])
>>> extracted_payload = ExtractedPayload(b'this is a payload with bad stuff', meta)
```

Should Scan

Likewise, there may be cases where an extracted payload should not be scanned by workers, but should be added to the results or archived. Simply set *PayloadMeta.should_scan* to *False*.

```
>>> meta = PayloadMeta(should_scan=False)
>>> extracted_payload = ExtractedPayload(b'this is a payload', meta)
```

API

class stoq.plugins.worker.**WorkerPlugin**(*config*)

abstract async scan(*payload, request*)

Return type Optional[*WorkerResponse*]

Response

class stoq.data_classes.**WorkerResponse**(*results=None, extracted=None, errors=None, dispatch_to=None*)

Object containing response from worker plugins

Parameters

- **results** (Optional[Dict]) – Results from worker scan
- **extracted** (Optional[List[ExtractedPayload]]) – ExtractedPayload objects of extracted payloads from scan
- **errors** (Optional[List[Error]]) – Errors that occurred

```
>>> from stoq import WorkerResponse, ExtractedPayload
>>> results = {'is_bad': True, 'filetype': 'executable'}
>>> extracted_payload = [ExtractedPayload(content=data, payload_meta=extracted_
↪meta)]
>>> response = WorkerResponse(results=results, extracted=extracted_payload)
```

Upgrading Plugins

v2 to v3

With the release of stoQ v3, a few enhancements were introduced that requires v2 plugins be slightly modified for use with v3. Some key changes include:

- Full asyncio support with all plugins
- The entire request state is passed to dispatchers, workers, and archivers. This includes making all payloads, and their respective results, available to them.
- A `Logger` object is now available to all plugins upon instantiation
- Errors from plugins are no longer simply a list of strings, they are now a list of `Error` objects
- Configuration parameters are passed to each plugin as a `StoqConfigParser` object rather than a `ConfigParser` object

- DeepDispatcher plugins have been deprecated

__init__

All plugin classes are instantiated exactly the same way. If the plugin requires additional configuration options, the `__init__` function may be added to your plugin class.

Key Changes:

- `from configparser import ConfigParser` has been replaced with a helper function and should be imported as `from stoq.helpers import StoqConfigParser`
- `plugins_opts` has been deprecated. All plugin options are now available within the `config` argument. `plugins_opts` must be removed from the `__init__` signature as well as from `super().__init__`

v2

```
from typing import Dict, Optional
from configparser import ConfigParser

class MyPlugin(ConnectorPlugin):
    def __init__(self, config: ConfigParser, plugin_opts: Optional[Dict]) -> None:
        super().__init__(config, plugin_opts)

        if plugin_opts and 'my_setting' in plugin_opts:
            self.my_setting = plugin_opts['my_setting']
        elif config.has_option('options', 'my_setting'):
            self.my_setting = config.get('options', 'my_setting')
        else:
            self.my_setting = None
```

v3

```
from stoq.helpers import StoqConfigParser


class MyPlugin(ConnectorPlugin):
    def __init__(self, config: StoqConfigParser) -> None:
        super().__init__(config)

        self.my_setting = config.get('options', 'my_setting', fallback=None)
```

ArchiverPlugin

Key Updates:

- import of `RequestMeta` is replaced with `Request`
- The `archive` function signature accepts a `Request` object rather than `RequestMeta`
- `def archive` is an async function, and must be changed to `async def archive`
- `def get` is an async function, and must be changed to `async def get`

v2

```
from stoq.plugins import ArchiverPlugin
from stoq import Payload, RequestMeta, ArchiverResponse


class MyArchiver(ArchiverPlugin):
    def archive(
        self, payload: Payload, request_meta: RequestMeta
    ) -> ArchiverResponse
        return ArchiverResponse

    def get(self, task: ArchiverResponse) -> Payload:
        return Payload()
```

v3

```
from stoq.plugins import ArchiverPlugin
from stoq import Payload, RequestMeta, ArchiverResponse


class MyArchiver(ArchiverPlugin):
    async def archive(
        self, payload: Payload, request: Request
    ) -> ArchiverResponse
        return ArchiverResponse

    async def get(self, task: ArchiverResponse) -> Payload:
        return Payload()
```

ConnectorPlugin

Key Updates:

- `def save` is an async function, and must be changed to `async def save`

v2

```
from stoq.plugins import ConnectorPlugin
from stoq import StoqResponse


class MyConnector(ConnectorPlugin):
    def save(self, response: StoqResponse) -> None:
        print(f'saving: {response}')
```

v3

```python
from stoq.plugins import ConnectorPlugin
from stoq import StoqResponse


class MyConnector(ConnectorPlugin):
    async def save(self, response: StoqResponse) -> None:
        print(f'saving: {response}')
```

DecoratorPlugin

Key Updates:

- `def decorate` is an async function, and must be changed to `async def decorate`

v2

```python
from stoq.plugins import DecoratorPlugin
from stoq import StoqResponse, DecoratorResponse


class MyDecorator(DecoratorPlugin):
    def decorate(self, response: StoqResponse) -> DecoratorResponse:
        return DecoratorResponse()
```

v3

```python
from stoq.plugins import DecoratorPlugin
from stoq import StoqResponse, DecoratorResponse


class MyDecorator(DecoratorPlugin):
    async def decorate(self, response: StoqResponse) -> DecoratorResponse:
        return DecoratorResponse()
```

DispatcherPlugin

Key Updates:

- import of `RequestMeta` is replaced with `Request`
- The `get_dispatches` function signature accepts a `Request` object rather than `RequestMeta`
- `def get_dispatches` is an async function, and must be changed to `async def get_dispatches`

v2

```
from stoq.plugins import DispatcherPlugin
from stoq import Payload, RequestMeta, DispatcherResponse


class MyDispatcher(DispatcherPlugin):
    def get_dispatches(
        self, payload: Payload, request_meta: RequestMeta
    ) -> DispatcherResponse:
        return DispatcherResponse()
```

v3

```
from stoq.plugins import DispatcherPlugin
from stoq import Payload, Request, DispatcherResponse


class MyDispatcher(DispatcherPlugin):
    async def get_dispatches(
        self, payload: Payload, request: Request
    ) -> DispatcherResponse:
        return DispatcherResponse()
```

ProviderPlugin

Key Updates:

- `from queue import Queue` is replaced with `from asyncio import Queue`
- `def ingest` is an async function, and must be changed to `async def ingest`
- When placing objects on the `Queue`, the call must be awaited, `await queue.put()`

v2

```
from queue import Queue
from stoq.plugins import ProviderPlugin
from stoq import Payload


class MyProvider(ProviderPlugin):
    def ingest(self, queue: Queue) -> None:
        queue.put(Payload(b'This is my payload'))
```

v3

```python
from asyncio import Queue
from stoq.plugins import ProviderPlugin
from stoq import Payload


class MyProvider(ProviderPlugin):
    async def ingest(self, queue: Queue) -> None:
        await queue.put(Payload(b'This is my payload'))
```

WorkerPlugin

Key Updates:

- import of `RequestMeta` is replaced with `Request`
- The `scan` function signature accepts a `Request` object rather than `RequestMeta`
- `def scan` is an async function, and must be changed to `async def scan`

v2

```python
from stoq.plugins import WorkerPlugin
from stoq import Payload, RequestMeta, WorkerResponse


class MyWorker(WorkerPlugin):
    def scan(self, payload: Payload, request_meta: RequestMeta) -> WorkerResponse:
        return WorkerResponse()
```

v3

```python
from stoq.plugins import WorkerPlugin
from stoq import Payload, Request, WorkerResponse


class MyWorker(WorkerPlugin):
    async def scan(self, payload: Payload, request: Request) -> WorkerResponse:
        return WorkerResponse()
```

Packaging Plugins

stoQ has a built-in plugin installation and upgrade capability. *stoQ* plugins may be packaged to allow for a simple and consistent installation process. Though packaging plugins isn't a necessity, it is highly recommended to do so for simplicity and reproducibility.

Let's take a look at a basic directory structure for a *stoQ* plugin:

```
|-- example_plugin/
|   `-- setup.py
|   `-- MANIFEST.in
|   `-- requirements.txt
|   `-- example_plugin/
|       `-- __init__.py
|       `-- example_plugin.py
|       `-- example_plugin.stoq
```

stoQ plugin packages leverage python's packaging library, setuptools. When a plugin is installed, `pip` is used for package management and installation. As such, all rules for both apply for *stoQ* plugins.

setup.py

The setup.py file is a standard `setuptools` script. `include_package_data` should always be set to `True` to ensure the plugin configuration file and any additional files are properly installed.

```python
from setuptools import setup, find_packages
setup(
    name="example_plugin",
    version="3.0.0",
    author="Marcus LaFerrera (@mlaferrera)",
    url="https://github.com/PUNCH-Cyber/stoq-plugins-public",
    license="Apache License 2.0",
    description="Example stoQ plugin",
    packages=find_packages(),
    include_package_data=True,
)
```

MANIFEST.in

The manifest file ensure that the plugins `.stoq` configuration file, and any other required files, are installed alongside the plugin. More information on the `.stoq` configuration file *can be found here*.

```
include example_plugin/*.stoq
```

requirements.txt

If a requirements file exists, *stoQ* will install dependencies appropriately. They will not be installed along side the plugin, but rather in python's system path. This file is not required if no additional dependencies need to be installed.

plugin subdirectory

The subdirectory above, `example_plugin`, is the primary plugin directory. This is the core location for the *stoQ* plugin that will be installed into the *stoQ* plugin directory. The plugin module, along with files identified in *MANIFEST.in* will be copied.

More information on writing a plugin *can be found here*.

Examples

There are plenty of examples for packaging plugin in *stoQ's* public plugin repository.

Frequently Asked Questions

- **What is the difference between stoQ v2 and v3?**

  The basic workflow and concept between the two versions are nearly similar, but under the hood a lot has changed. Version 2 of stoQ was a complete rewrite of v1, filled with lots of lessons learned, optimizations, and best practices. Additionally, we made the decision to ensure a modern version of python was used in order to leverage many of the added benefits and features.

  stoQ v3 built upon v2, but added many additional features such as native AsyncIO support, streamlined data flow, and passing the full request stte to each worker plugin. A full list of changes can be found in the CHANGELOG.

- **Are plugins from v2 compatiable with v3?**

  Unfortunately, no. However, porting plugins to v3 is very simple. You can read more about that *here*.

- **Is v1 or v2 of stoQ still available?**

  Absolutely, though they are no longer maintained (minus major bug fixes or security issues) in favor of v3.

- **Why should I use stoQ?**

  Because your time is valuable and there are better things to do with it than run the same tools over and over again. stoQ allows you to automate most of the mundane tasks analysts do on a daily basis. It also allows you to do this scanning at scale, against a few to hundreds of millions of payloads daily.

- **How long has stoQ been around?**

  We started developing stoQ back in 2011 to help automate and streamline many of our day to day tasks. In 2015, after several years of developing and real world use in large enterprise environments, we decided to open source the entire framework, along with many plugins.

- **Why is everything a plugin?**

  Flexibility. When we started building stoQ we didn't want to have to reengineer it if we switched databases, or if we wanted to use a different queuing system, or some other random piece of our workflow changed. By leveraging plugins, it's simple a matter of adding or removing them.

- **Can stoQ be used for more than just file analysis?**

  Absolutely. We've used it for everything from processing threat intel feeds, to scanning e-mails (and their attachments), to slack bots.

- **How does stoQ work at scale?**

  As with anything that "scales", it depends. Infrastructure, location, resources, and many other things come into play. In our experience, it is possible to scan hundreds of millions of payloads per day with the right setup. Overall, we have been very pleased with it's ability to scale to fit all of our needs without issue.

- **stoQ seems slow when decoding json, can this be improved?**

  Possibly. stoQ leverages BeautifulSoup's UnicodeDammit function to serialize bytes into proper json serializable content. In order to limit the python library requirements and maximize compatibility, we purposefully limit core dependencies. BeautifulSoup by default attempts to leverage the python library *cchardet*, which is much more efficient than the default python library that BeautifulSoup falls back to *chardet*. Simply install *cchardet* via pip, and you may see a nice performance boost if you have complex results with bytes.

- **I know stoQ supports async operations, but my plugins don't seem to be completing any faster!**

  While all current stoQ plugins support the latest version of stoQ, not all of them will run asynchronously. There are several reasons for this. Some depend on 3rd party libraries that are not asyncio compatiable. For these, we will keep an eye out for updated 3rd party libraries that support asyncio. For many others, it is simply a matter of competing priorities. We, and very gratefully, several contributors to stoQ have been updating plugins for full asyncio support, it is still a time consuming process. If you would like to help in this effort, please do! We are more than happy to accept all of the help you are willing to volunteer.

- **Do you plan on maintaining this project long term?**

  Absolutely. We use stoQ in several production grade capabilities, as do many stoQ users. We've been developing it since 2011, and will continue to do so.

- **Can I contribute?**

  Of course! Check out the *contributing section* to find out how.

- **Something seems broken, how can I get help?**

  Feel free to submit an issue.

- **How can I ask other questions?**

  Feel free to join us on spectrum, reach out to us at @punchcyber or the author @mlaferrera

1.6.2 Community Guide

Community Guide

Looking at learning more about the project and how to contribute? Read on.

Contributing

Welcome

Thank you for considering contributing to stoQ. It's people like you that keep projects relevant and useful for the community. Our team looks forward to collaborating with you.

How to Contribute

There are many ways to contribute, especially since stoQ is a full stack project. Our team is small so any contributions are more than welcome :) That includes tutorials, posts, documentation improvements, bug reports, feature requests, etc.

Ground Rules

Expectations

Keep in mind that repository maintainers and community contributors for stoQ are volunteers. Respect amongst participants must be maintained at all times and collaboration should be constructive. Everyone is expected to act in accordance with the project's *Code of Conduct*

How to report a bug

Security Disclosures

If you find a security vulnerability, do NOT open an issue. Email info @ punchcyber.com instead.

Other Bug Disclosures

For other bugs, please open an issue on our issues list on Github and include as much information as possible.

Suggest Features or Enhancements

If you have an idea for a feature that doesn't exist in stoQ, there are likely others out there with a similar need. Open an issue on our issues list on GitHub which describes the feature you would like to see, why you need it, and how it should work.

Code review process

Contribution Acceptance

Our team is small so Pull Requests are reviewed as time allows. Depending on the size of the request this may take some time. However, request submitters can expect initial comments or a status update from our team within two weeks. Please be patient :)

Code of Conduct

Simply put, just be nice.

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at info@punchcyber.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the [Contributor Covenant][homepage], version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

[homepage]: https://www.contributor-covenant.org

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX