BroAPT

Release 2020.03.14

Jarry Shaw

Aug 09, 2023

CONTENTS

CHAPTER

ONE

QUICKSTART

1.1 Installation

Installation of the BroAPT system is rather simple. Just clone the repository or download the tarball, then voilà, it's ready to go.

```
# from GitHub (active repository)
git clone https://github.com/JarryShaw/BroAPT.git
# or from GitLab (authentication required)
git clone https://gitlab.sjtu.edu.cn/bysj/2019bysj.git
```

1.2 Usage

1.2.1 `broaptd` Service

On Linux systems, you can register a System V service for `broaptd`, the main entrypoint of the BroAPT system, a.k.a the CLI of BroAPT-Daemon server.

Important: We suppose you're installing `broaptd` on a CentOS or similar distribution. For macOS binaries and Docker Compose, you may find them with `darwin` suffix.

For macOS services, you can register through the Launch Agent of macOS system. See *launchd(8)* and *launchd.plist(5)* for more information.

---

0. Install the `broaptd` binary:

   ```
   # from bundled implementation
   sudo cp source/server/bin/broapt.linux /usr/local/bin/broaptd
   # from cluster implementation
   sudo cp cluster/daemon/bin/broapt.linux /usr/local/bin/broaptd
   ```

   The binary is built using `PyInstaller`. Should you wish to build a suitable binary for your target system, please refer to the `.spec` files at `source/server/spec/` (for bundled implementation) or `cluster/daemon/spec/` (for cluster implementation).

1. Create a *dotenv* file named `/etc/sysconfig/broaptd`:

```
## daemon kill signal
BROAPT_KILL_SIGNAL=15  # TERM

## BroAPT-Daemon server
BROAPT_SERVER_HOST="127.0.0.1"
BROAPT_SERVER_PORT=5000

## path to BroAPT's docker-compose.yml
# for bundled implementation
BROAPT_DOCKER_COMPOSE="/path/to/broapt/source/docker/docker-compose.linux.yml"
# for cluster implementation
BROAPT_DOCKER_COMPOSE="/path/to/broapt/cluster/docker/docker-compose.linux.yml"

## path to extract files
BROAPT_DUMP_PATH="/path/to/extract/file/"
## path to log files
BROAPT_LOGS_PATH="/path/to/log/bro/"
## path to detection APIs
# for bundled implementation
BROAPT_API_ROOT="/path/to/broapt/source/client/include/api/"
# for cluster implementation
BROAPT_API_ROOT="/path/to/broapt/cluster/app/include/api/"
## path to API runtime logs
BROAPT_API_LOGS="/path/to/log/bro/api/"

## sleep interval
BROAPT_INTERVAL=10
## command retry
BROAPT_MAX_RETRY=3
```

2. Create a System V service file at `/etc/systemd/system/broaptd.service` (works on Ubuntu 18.04):

```
[Unit]
Description=BroAPT Daemon

[Service]
ExecStart=/usr/local/bin/broaptd --env /etc/sysconfig/broaptd
ExecReload=/usr/bin/kill -INT $MAINPID
Restart=always
RestartSec=60s

[Install]
WantedBy=multi-user.target
```

3. Reload daemon and enable `broaptd` service:

```
sudo systemctl daemon-reload
sudo systemctl enable broaptd.service
```

You may wish to check if its running now:

```
sudo systemctl status broaptd.service
```

1.2.2 Docker Image

The BroAPT Docker images can be found on Docker Hub now.

- Bundled implementation: `jsnbzh/broapt:latest`
- Cluster implementation:
  - BroAPT-Core framework: `jsnbzh/broapt:core`
  - BroAPT-App framework: `jsnbzh/broapt:app`

1.2.3 Docker Compose

Even though the `broaptd` will already manage the Docker containers of the BroAPT system through Docker Compose, you might wish to check by yourself.

Bundled Implementation

For bundled implementation, there is only one Docker container service called `broapt`. You can refer to the Docker Compose file at `source/docker/docker-compose.${system}.yml`.

Cluster Implementation

For cluster implementation, there are two Docker container services: `core` for the BroAPT-Core framework and `app` for the BroAPT-App framework. You can refer to the Docker Compose file at `cluster/docker/docker-compose.${system}.yml`.

1.3 Repository Structure

```
/broapt/
├── LICENSE             # CC license
├── LICENSE.bsd         # BSD license
├── cluster             # cluster (standalone) implementation
│   └── ...
├── docs
│   ├── broaptd.8       # manual for BroAPT-Daemon
│   ├── thesis.pdf      # Bachelor's Thesis
│   └── ...
├── gitlab              # GitLab submodule
│   └── ...
├── source              # bundled (all-in-one) implementation
│   └── ...
├── vendor              # vendors, archives & dependencies
│   └── ...
└── ...
```

CHAPTER

TWO

CONFIGURATIONS

As discussed in previous sections, the BroAPT system is configurable in various ways. You can configure the outer system from the entry CLI of BroAPT-Daemon server, and the main framework through Docker Compose environment variables.

2.1 BroAPT-Daemon Server

2.1.1 Command Line Interface

```
usage: broaptd [-h] [-v] [-e ENV] [-s SIGNAL] [-t HOST] [-p PORT]
               [-f DOCKER_COMPOSE] [-d DUMP_PATH] [-l LOGS_PATH] [-r API_ROOT]
               [-a API_LOGS] [-i INTERVAL] [-m MAX_RETRY]

BroAPT Daemon

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit

environment arguments:
  -e ENV, --env ENV     path to dotenv file
  -s SIGNAL, --signal SIGNAL
                        daemon kill signal

server arguments:
  -t HOST, --host HOST  the hostname to listen on
  -p PORT, --port PORT  the port of the webserver

compose arguments:
  -f DOCKER_COMPOSE, --docker-compose DOCKER_COMPOSE
                        path to BroAPT's compose file
  -d DUMP_PATH, --dump-path DUMP_PATH
                        path to extracted files
  -l LOGS_PATH, --logs-path LOGS_PATH
                        path to log files

API arguments:
  -r API_ROOT, --api-root API_ROOT
                        path to detection APIs
```

(continues on next page)

(continued from previous page)

```
  -a API_LOGS, --api-logs API_LOGS
                        path to API runtime logs

runtime arguments:
  -i INTERVAL, --interval INTERVAL
                        sleep interval
  -m MAX_RETRY, --max-retry MAX_RETRY
                        command retry
```

2.1.2 Environment Variables

As suggests in the `--env` option, you may provice a *dotenv* (`.env`) file for the BroAPT-Daemon server to configure itself.

Acceptable environment variables are as following:

BROAPT_KILL_SIGNAL

Type
`int`

Default
`15` (`SIGTERM`)

CLI Option
`-s` / `--signal`

Daemon kill signal.

BROAPT_SERVER_HOST

Type
`str` (hostname)

Default
`0.0.0.0`

CLI Option
`-t` / `--host`

The hostname to listen on.

BROAPT_SERVER_PORT

Type
`int` (port number)

Default
`5000`

CLI Option
`-p` / `--port`

The port of the webserver.

BROAPT_DOCKER_COMPOSE

Type
`str` (path)

Default
`docker-compose.yml`

CLI Option
`-f` / `--docker-compose`

Path to BroAPT's compose file.

BROAPT_DUMP_PATH

Type
`str` (path)

Default
`None`

CLI Option
`-d` / `--dump-path`

Path to extracted files.

BROAPT_LOGS_PATH

Type
`str` (path)

Default
`None`

CLI Option
`-l` / `--logs-path`

Path to log files.

BROAPT_API_ROOT

Type
`str` (path)

Default
`None`

CLI Option
`-r` / `--api-root`

Path to detection APIs.

BROAPT_API_LOGS

Type
`str` (path)

Default
`None`

CLI Option
`-a` / `--api-logs`

Path to API runtime logs.

BROAPT_INTERVAL

Type
`float`

Default
10

CLI Option
-i / --interval

Sleep interval.

BROAPT_MAX_RETRY

Type
int

Default
3

CLI Option
-m / --max-retry

Command retry.

Note: Environment variables of bool type will be translated through the following mapping table (**case-insensitive**):

True	False
1	0
yes	no
true	false
on	off

2.2 BroAPT-Core Framework

The BroAPT-Core framework only supports configuration through environment variables.

BROAPT_CPU

Type
int

Default
None

Availability
bundled implementation

Number of BroAPT concurrent processes for PCAP analysis. If not provided, then the number of system CPUs will be used.

BROAPT_CORE_CPU

Type
int

Default
None

Availability
cluster implementation

See also:

BROAPT_CPU

BROAPT_INTERVAL

Type
`float`

Default
`10`

Availability
bundled implementation

Wait interval after processing current pool.

BROAPT_CORE_INTERVAL

Type
`float`

Default
`10`

Availability
cluster implementation

Wait interval after processing current pool of PCAP files.

BROAPT_DUMP_PATH

Type
`str` (path)

Default
`FileExtract::prefix` (Bro script)

Path to extracted files.

BROAPT_PCAP_PATH

Type
`str` (path)

Default
`/pcap/`

Path to source PCAP files.

BROAPT_LOGS_PATH

Type
`str` (path)

Default
`/var/log/bro/`

Path to system logs.

BROAPT_MIME_MODE

Type
`bool`

Default
`True`

If group extracted files by MIME type.

BROAPT_JSON_MODE

Type
`bool`

Default
`LogAscii::use_json` (Bro script)

Toggle Bro logs in JSON or ASCII format.

BROAPT_BARE_MODE

Type
`bool`

Default
`False`

Run Bro in bare mode (don't load scripts from the `base/` directory).

BROAPT_NO_CHKSUM

Type
`bool`

Default
`True`

Ignore checksums of packets in PCAP files when running Bro.

BROAPT_HASH_MD5

Type
`bool`

Default
`False`

Calculate MD5 hash of extracted files.

BROAPT_HASH_SHA1

Type
`bool`

Default
`False`

Calculate SHA1 hash of extracted files.

BROAPT_HASH_SHA256

Type
`bool`

Default
`False`

Calculate SHA256 hash of extracted files.

`BROAPT_X509_MODE`

Type
`bool`

Default
`False`

Include X509 information when running Bro.

`BROAPT_ENTROPY_MODE`

Type
`bool`

Default
`False`

Include file entropy information when running Bro.

`BROAPT_LOAD_MIME`

Type
`List[str]` (*case-insensitive*)

Default
`None`

A `,` or `;` separated string of MIME types to be extracted.

`BROAPT_LOAD_PROTOCOL`

Type
`List[str]` (*case-insensitive*)

Default
`None`

A `,` or `;` separated string of application layer protocols to be extracted, can be any of `dtls`, `ftp`, `http`, `irc` and `smtp`.

`BROAPT_FILE_BUFFER`

Type
`int` (`uint64`)

Default
`Files::reassembly_buffer_size` (Bro script)

Reassembly buffer size for file extraction.

`BROAPT_SIZE_LIMIT`

Type
`int` (`uint64`)

Default
`FileExtract::default_limit` (Bro script)

Size limit of extracted files.

BROAPT_HOOK_CPU

Type
`int`

Default
`1`

Number of BroAPT concurrent processes for Python hooks.

2.3 BroAPT-App Framework

The BroAPT-App framework only supports configuration through environment variables.

BROAPT_SCAN_CPU

Type
`int`

Default
`None`

Availability
bundled implementation

Number of BroAPT concurrent processes for extracted file analysis. If not provided, then the number of system CPUs will be used.

BROAPT_APP_CPU

Type
`int`

Default
`None`

Availability
cluster implementation

See also:

BROAPT_SCAN_CPU

BROAPT_INTERVAL

Type
`float`

Default
`10`

Availability
bundled implementation

Wait interval after processing current pool.

BROAPT_APP_INTERVAL

Type
`float`

Default
10

Availability
cluster implementation

Wait interval after processing current pool of extracted files.

BROAPT_MAX_RETRY

Type
int

Default
3

Retry times for failed commands.

BROAPT_API_ROOT

Type
str (path)

Default
/api/

Path to the API root folder.

BROAPT_API_LOGS

Type
str (path)

Default
/var/log/bro/api/

Path to API detection logs.

BROAPT_NAME_HOST

Type
str (hostname)

Default
localhost

Hostname of BroAPT-Daemon server.

BROAPT_NAME_PORT

Type
int (port number)

Default
5000

Port number of BroAPT-Daemon server.

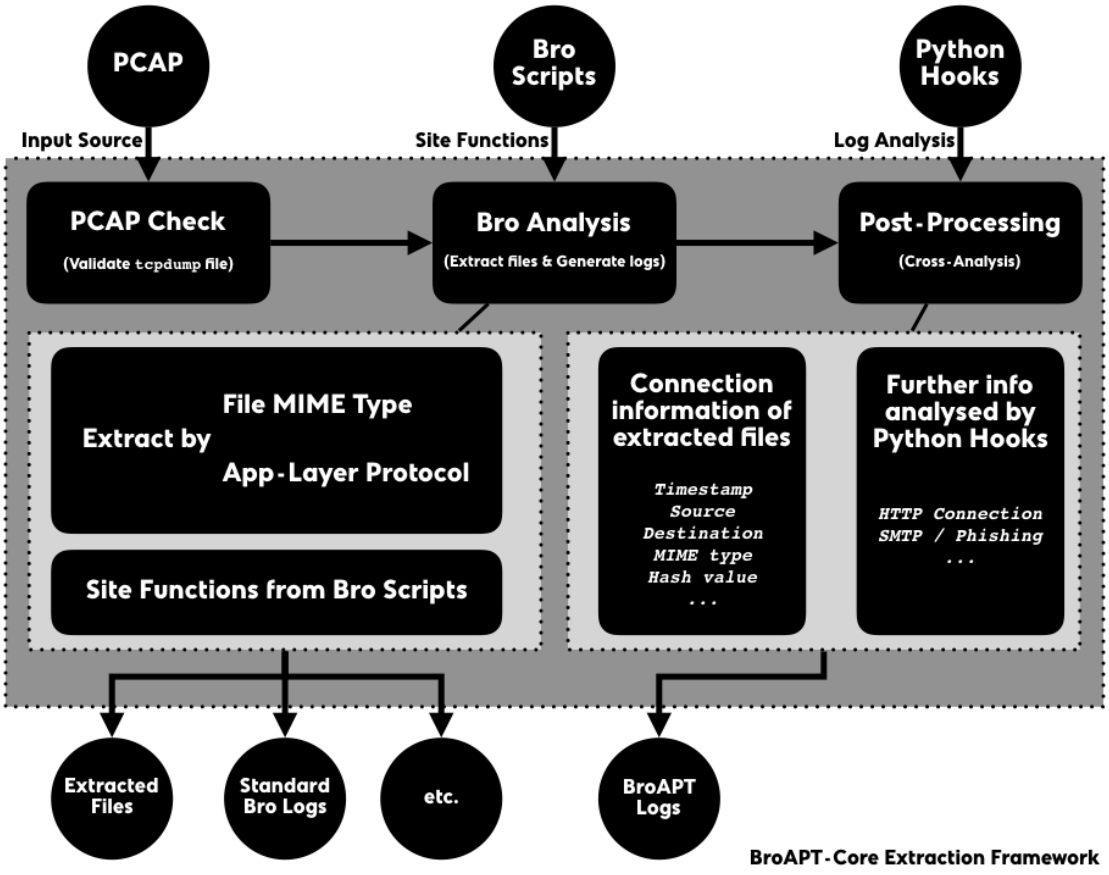
CHAPTER

THREE

INTERNAL FRAMEWORKS

3.1 BroAPT-Core Extration Framework

The BroAPT-Core framework processes PCAP files, extracts files transferred through traffic contained in the PCAP files, and perform analysis to the log files generated by Bro scripts.



0. When the BroAPT-Core framework first reads in a new PCAP file, it will validate if it's a valid `tcpdump` (*`tcpdump(1)`*) format file, through `libmagic` (*`libmagic(3)`*).
1. If validated, the BroAPT-Core framework will utilise the Bro IDS to perform analysis upon the PCAP file, extracting files and generating logs.

When extracting, you may toggle through *environment variables* to configure which MIME types and/or what application layer protocol files transferred with should be extracted.

Also, site functions from user-defined Bro scripts will be loaded and executed at the same time.

This step will produce extracted files and standard Bro logs, as well as extra artefacts elevated through the site functions.

2. Later, the BroAPT-Core framework will perform post-processing, a.k.a. cross-analysis, upon the logs generated in previous step.

   By default, the BroAPT-Core framework will gather connection information of the extracted files from the Bro logs (`files.log`). Some other analysis will also be performed as defined in the Python hooks.

   The result of analysis will be elevated as BroAPT logs.

3.1.1 Custom Bro Scripts

In the BroAPT system, you can customise your own Bro script. The BroAPT-Core framework will load those scripts when running Bro IDS to process PCAP files.

User defined Bro scripts will be mapped into the Docker container at runtime. The directory structure would be as following:

```
/broapt/scripts/
│   # load FileExtraction module
├── __load__.bro
│   # configurations
├── config.bro
│   # MIME-extension mappings
├── file-extensions.bro
│   # protocol hooks
├── hooks/
│   │   # extract DTLS
│   ├── extract-dtls.bro
│   │   # extract FTP_DATA
│   ├── extract-ftp.bro
│   │   # extract HTTP
│   ├── extract-http.bro
│   │   # extract IRC_DATA
│   ├── extract-irc.bro
│   │   # extract SMTP
│   └── extract-smtp.bro
│   # core logic
├── main.bro
│   # MIME hooks
├── plugins/
│   │   # extract all files
│   ├── extract-all-files.bro
│   │   # extract APK
│   ├── extract-application-vnd-android-package-archive.bro
│   │   # extract PDF
│   ├── extract-application-pdf.bro
│   │   # extract PE
│   ├── extract-application-vnd-microsoft-portable-executable.bro
```

(continues on next page)

(continued from previous page)

```
│   │   # extract by BRO_MIME
│   └── extract-white-list.bro
│   # site functions by user
└── sites/
    │   # load site functions
    ├── __load__.bro
    └── ...
```

where `extract-application-vnd-android-package-archive.bro`, `extract-application-pdf.bro` and `extract-application-vnd-microsoft-portable-executable.bro` are Bro scripts generated automatically by the BroAPT-Core framework based on the *BROAPT_LOAD_MIME* environment vairable.

Important: The *BROAPT_LOAD_MIME* supports UNIX *shell*-like pattern matching, c.f. `fnmatch` module from Python.

And `/broapt/scripts/sites/` are mapped from the host machine, which includes the Bro scripts defined by user. You may include your scripts into the BroAPT-Core framework by loading (`@load`) them in the `/broapt/scripts/sites/__load__.bro` file.

At the moment, we have six sets of Bro scripts included in the distribution.

Common Constants

In the BroAPT system, it predefines many constants of common protocols and systems, such as FTP commands, HTTP methods, etc. We used crawlers to fetch relevant data from the IANA registry, generate and/or update Bro constants, such as `HTTP::header_names` for HTTP headers fields.

HTTP Cookies

The script utilised `http_header` event, and extends the builtin `http.log` record object `HTTP::Info` with data from the `COOKIE` header.

Unknown HTTP Headers

As defined in **RFC 2616** and **RFC 7230**, and registered in IANA, there're a list of known HTTP headers. However, customised headers may be introduced when implementation. Such unknown headers may contain significant information about the HTTP traffic. Therefore, the script utilised `http_header` event and search for unknown headers, i.e. not included in `HTTP::header_names`, then record them in the `http.log` files.

HTTP `POST` Data

As **RFC 2616** suggests, we can utilise the data sent from `POST` command to analyse information about outbound traffic. The script utilised `http_entity_data` event, and save the `POST` data to `http.log` files.

Calculate Hash Values

Hash value of files can be used to detect malware. The script utilised `file_new` event, calculated and saved the hash values of files transferred in the `files.log` file.

SMTP Phishing Detect

Since files transferred through SMTP traffic are not easy to gather and detect phishing information. We introduced two Bro modules to perform such detection on the SMTP traffic.

A. `Phishing` Module

The `Phishing` module mainly provides mass scam emails; phishing email detection based on Levenshtein distance of sender address. It will elevate a `phishing_link.log` log file, containing such malicious connections and URLs.

B. `Phish` Module

Primary scope of these bro policies is to give more insights into smtp-analysis esp to track phishing events.

This is a subset of phish-analysis repo and doesn't use any backed `postgres` database. So relieves the user from `postgres` dependency while getting basic phishing detection up and running very quickly.

3.1.2 Custom Python Hooks

In the BroAPT system, you can customise your own Python hooks for cross-analysis to the log files. The BroAPT-Core framework will call such registered hooks on each set of log files generacted from a PCAP file after processing of Bro.

See also:

Log analysis and generation can be done through the ZLogging project, which provides both loading and dumping interface to the processing of Bro logs in an elegant Pythonic way.

User defined Bro scripts will be mapped into the Docker container at runtime. The directory structure would be as following:

```
/broapt/python/
    # setup PYTHONPATH
├── __init__.py
    # entry point
├── __main__.py
    # config parser
├── cfgparser.py
    # Bro script composer
├── compose.py
    # global constants
├── const.py
    # Bro log parser
├── logparser.py
    # BroAPT-Core logic
├── process.py
    # multiprocessing support
├── remote.py
```

(continues on next page)

(continued from previous page)

```
    # BroAPT-App logic
├── scan.py
    # Python hooks
├── sites
    │   # register hooks
    ├── __init__.py
    └── ...
    # utility functions
└── utils.py
```

where `/broapt/python/sites/` is mapped from the host machine, which includes user-defined site customisation Python hooks.

You can register your own hooks in the `/broapt/python/sites/__init__.py`, by importing (`import`) them and add them to the `HOOK` and/or `EXIT` registry lists.

In the `HOOK` registry, each registered hook function will be called after a PCAP file is processed by the Bro IDS, and perform analysis on the logs generated from the PCAP file.

Note: The hook function will be called with **ONE** argument, `log_name`, a string (`str`) representing the folder name to the target logs.

In the `EXIT` registry, each registered hook function will be called before the main process of the BroAPT-Core framework exits.

Note: The hook function will be called with **NO** argument.

At the moment, we have bundled two sets of Python hooks in the system.

Extracted File Information

Through `conn.log` and `files.log`, the BroAPT system generates a new log file for information of extracted files, which includes the timestamp, source and destination IP addresses of the transport layer connection (TCP/UDP) transferring the file, MIME type of the file, as well as hash values, see below:

Field Name	Bro Type	Description
`timestamp`	`float`	Connection timestamp
`log_uuid`	`string`	UUID of source logs
`log_path`	`string`	Absolute path to source logs (in Docker container)
`log_name`	`string`	Relative path to source logs
`dump_path`	`string`	Absolute path to extracted file (in Docker container)
`local_name`	`string`	Relative path to extracted file
`source_name`	`string`	Original filename (if present)
`hosts`	`vector`	Transferrer and receiver
`conns`	`vector`	Source and destination IP addresses and ports
`bro_mime_type`	`string`	MIME type probed by Bro IDS
`real_mime_type`	`string`	MIME type detected by `libmagic`
`hash`	`table`	Hash values (MD5, SHA1 and SHA256)

The equivalent ZLogging data model can be declared as following:

```
class ExtractedFiles(Model):
    timestamp = FloatType()
    log_uuid = StringType()
    log_path = StringType()
    dump_path = StringType()
    local_name = StringType()
    source_name  = StringType()
    hosts = VectorType(element_type=RecordType(
        tx=AddrType(),
        rx=AddrType(),
    ))
    conns = VectorType(element_type=RecordType(
        src_h=AddrType(),
        src_p=PortType(),
        dst_h=AddrType(),
        dst_p=PortType(),
    ))
    bro_mime_type = StringType()
    real_mime_type = StringType()
    hash = RecordType(
        md5=StringType(),
        sha1=StringType(),
        sha256=StringType(),
    )
```

HTTP Connection Information

Through analysis upon `http.log`, the BroAPT system elevated a new log file with more concentrated information about HTTP connections. Such log file contains all HTTP connections from every processed PCAP file, and can be used for further analysis based on *big data*.

Field Name	Bro Type	Description
`srcip`	`addr`	Client IP address
`ts`	`float`	Request timestamp (microseconds)
`url`	`string`	Requests URL path
`ref`	`string`	`Referer` header of the request (*base64* encoded)
`ua`	`string`	`User-Agent` header of the request (*base64* encoded)
`dstip`	`addr`	Server IP address
`cookie`	`string`	`Cookie` header of the request (*base64* encoded)
`src_port`	`port`	Client port
`json`	`vector`	Unregistered HTTP header fields (*JSON* encoded)
`method`	`string`	HTTP method
`body`	`string`	`POST` body data (*base64* encoded)

The equivalent ZLogging data model can be declared as following (with type annotations):

```
class HTTPConnections(Model):
    srcip: bro_addr
    ts: bro_float
    url: bro_string
```

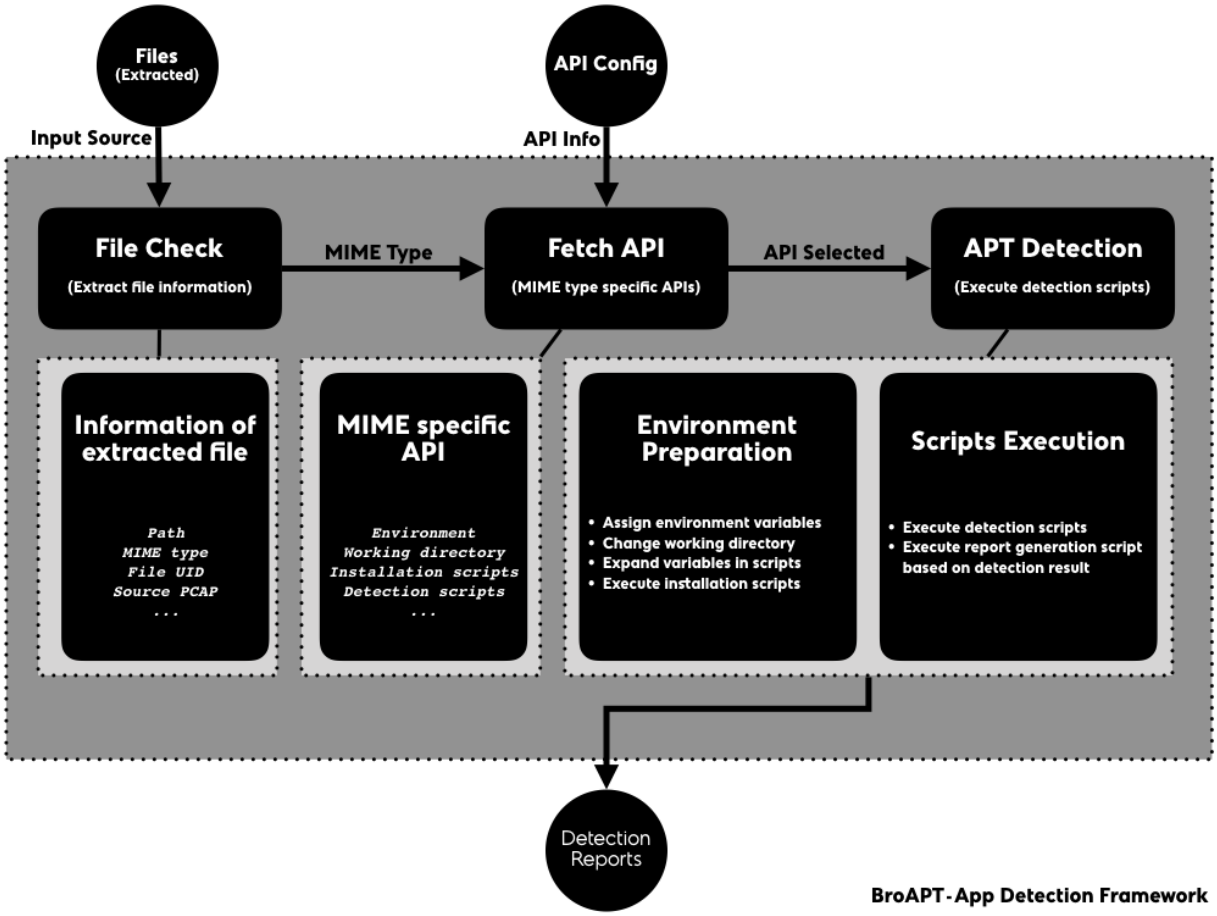
(continues on next page)

(continued from previous page)

```
    ref: bro_string
    ua: bro_string
    dstip: bro_addr
    cookie: bro_string
    src_port: bro_port
    json: bro_vector[bro_string]
    method: bro_string
    body: bro_string
```

3.2 BroAPT-App Detection Framework

The BroAPT-App framework processes extracted files, perform malware detection upon those files with detection API configured through the configuration file.



BroAPT-App Detection Framework

0. The BroAPT-App framework fetches basic information about the extracted file, including file path, MIME type, file UID, source PCAP file, etc.

   Each file extracted, since it will be named after:

   ```
   PROTOCOL-FUID-MIMETYPE.EXT
   ```

with such pattern, the BroAPT-App framework will generate an `Entry` to represent the information of the target file, e.g. for a extracted file named:

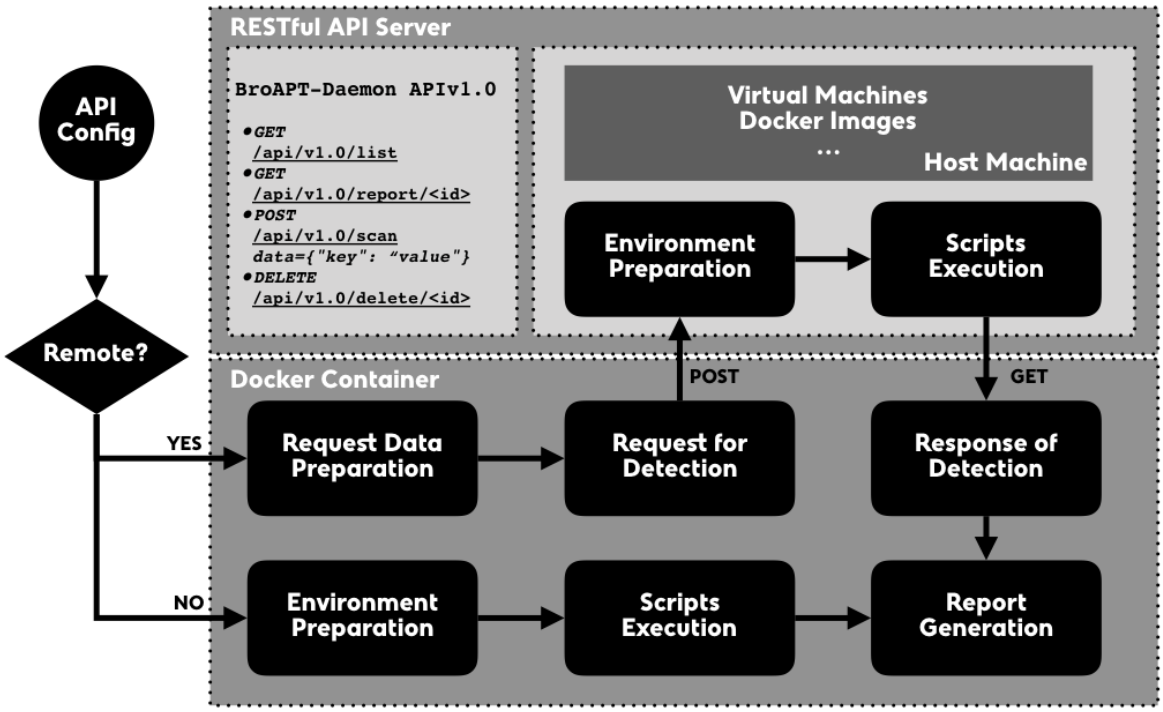
```
application/vnd.openxmlformats-officedocument/HTTP-F3Df5B3z9UI3yi5J03.application.
↪msword.docx
```

the BroAPT-App framework will generate the `Entry` object as following:

```
Entry(
    path='application/vnd.openxmlformats-officedocument/HTTP-F3Df5B3z9UI3yi5J03.
↪application.msword.docx',
    uuid='F3Df5B3z9UI3yi5J03',
    mime=MIME(
        media_type='application',
        subtype='msword',
        name='application/msword'
    )
)
```

1. Based on the MIME type, the BroAPT-App framework will obtain MIME specific detection API for the extracted file.
2. The BroAPT-App framework will then start detecting the extracted file based on the specification described in the API.

   When detection, as the Docker container may not be capable of such action, the BroAPT-App framework may request the BroAPT-Daemon server to *remote* detect the extracted file.



BroAPT Client-Server Detection Framework

The BroAPT-Daemon server is a RESTful API server implemented using Flask microframework. At the moment it supports following APIs:

URI Routing	HTTP Method	Description
`/api/v1.0/list`	`GET`	Query detection listing
`/api/v1.0/report/<id>`	`GET`	Query detection report
`/api/v1.0/scan data={"key": "value"}`	`POST`	Request *remote* detection
`/api/v1.0/delete/<id>`	`DELETE`	Delete detection record

3.2.1 MIME Specific API Configuration

In the BroAPT-App framework, we used an API configuration file to provide the BroAPT system with MIME specific detection mechanism. The configuration file is written in YAML, inspired by Docker Compose and Travis CI.

The directory structure of API configuration file and its related files are as below:

```
/api/
│   # API configuration file
├── api.yml
│   # MIME: application/*
├── application/
│   └── ...
│   # MIME: audio/*
├── audio/
│   └── ...
│   # default API
├── example/
│   └── ...
│   # MIME: font/*
├── font/
│   └── ...
│   # MIME: image/*
├── image/
│   └── ...
│   # MIME: message/*
├── message/
│   └── ...
│   # MIME: model/*
├── model/
│   └── ...
│   # MIME: multipart/*
├── multipart/
│   └── ...
│   # MIME: text/*
├── text/
│   └── ...
│   # MIME: video/*
└── video/
    └── ...
```

The `/api/` folder will be mapped into the Docker container at runtime and the `/api/api.yml` is the exact API configuration file. The API for `example` MIME type is the default fallback detection method for those with **NO** existing

detection API configured.

In the configuration file, you can specify global environment variables under the `environment` key:

```
environment:
  # API root path (from environment vairable)
  API_ROOT: ${BROAPT_API_ROOT}
  # Python 3.6
  PYTHON36: /usr/bin/python3.6
  # Python 2.7
  PYTHON27: /usr/bin/python
  # Shell/Bash
  SHELL: /bin/bash
```

And for a certain MIME, e.g. PDF files (MIME is `application/pdf`), the configuration should be as following:

```
application:
  pdf:
    remote: false
    # default working directory is ``/api/application/pdf/``
    # now changed to ``/api/application/pdf/pdf_analysis``
    workdir: pdf_analysis
    environ:
      ENV_FOO: 1
      ENV_BAR: cliche
    install:
      - apt-get update
      - apt-get install -y python python-pip
      - ${PYTHON27} -m pip install -r requirements.txt
      - rm -rf /var/lib/apt/lists/*
      - apt-get remove -y --auto-remove python-pip
      - apt-get clean
    scripts:
      - ${PYTHON27} detect.py [...]
      - ...
    report: ${PYTHON27} report.py
```

Note: *Shell*-like globing is now supported for MIME types, you may specify an API using `application/vnd.ms-*`, which will be used for both `application/vnd.ms-excel` and `application/ms-powerpoint`.

In the configuration file, the `report` key is mandatory.

If set `remote` key as `true`, the BroAPT-App framework will request the BroAPT-Daemon server to perform *remote* detection.

And if an API configuration is shared by multiple MIME types, you should set `shared` key as `true`, so that the API would be *process-safe* at runtime.

After parsing through the *`cfgparser.parse()`* function, the API configuration above will be represented as:

```
API(
    workdir='pdf_analysis',
    environ={
       'API_ROOT': '${BROAPT_API_ROOT}',
```

(continues on next page)

(continued from previous page)

```
        'PYTHON36': '/usr/bin/python3.6',
        'PYTHON27': '/usr/bin/python',
        'SHELL': '/bin/bash',
        'ENV_FOO': '1',
        'ENV_BAR': 'cliche'
    },
    install=[
        'apt-get update',
        'apt-get install -y python python-pip',
        '${PYTHON27} -m pip install -r requirements.txt',
        'rm -rf /var/lib/apt/lists/*',
        'apt-get remove -y --auto-remove python-pip',
        'apt-get clean'
    ],
    scripts=[
        '${PYTHON27} detect.py [...]',
        ...
    ],
    report='${PYTHON27} report.py',
    remote=False,
    shared='application/pdf',
    inited=<Synchronized wrapper for c_ubyte(0)>,
    locked=<Lock(owner=unknown)>
)
```

- `API.inited` is to mark if the installation process had been run successfully.
- `API.shared` is to mark if the configuration is shared by multiple MIME types.
- `API.locked` is to mark if the process is locked to prevent resource competition.

At runtime, if the BroAPT-App framework is to detect a file at `/dump/application/pdf/test.pdf`, the main procedure is as follows:

0. Set environment variables:

   ```
   API_ROOT="/api/"
   PYTHON36="/usr/bin/python3.6"
   PYTHON27="/usr/bin/python"
   SHELL="/bin/bash"
   ENV_FOO=1
   ENV_BAR="cliche"
   BROAPT_PATH="/dump/application/pdf/test.pdf"
   BROAPT_MIME="application/pdf"
   ```

1. Change the current working directory to `/api/application/pdf/pdf_analysis`.
2. If the `API.inited` is now `False`, which means the installation process is **NOT** yet performed, then acquire `API.locked` and execute the commands:

   ```
   apt-get update
   apt-get install -y python python-pip
   python -m pip install -r requirements.txt
   rm -rf /var/lib/apt/lists/*
   ```

(continues on next page)

(continued from previous page)

```
apt-get remove -y --auto-remove python-pip
apt-get clean
```

afterwards, toggle `API.inited` to `True` and release `API.locked`.

3. Execute detection commands:

```
/usr/bin/python detect.py [...]
...
```

4. Once finished, execute report generation script `/usr/bin/python report.py`.

3.2.2 Integrated Detection Services

At the moment, the BroAPT system had integrated six detection solusions.

Default Detection powered by VirusTotal

VirusTotal aggregates many antivirus products and online scan engines to check for viruses that the user's own antivirus may have missed, or to verify against any false positives.

As mentioned above, the `example` MIME type is the default fallback detection mechanism in case of missing configuration. The configuration is as below:

```
example:
  environ:
    ## sleep interval
    VT_INTERVAL: 30
    ## max retry for report
    VT_RETRY: 10
    ## percentage of positive threshold
    VT_PERCENT: 50
    ## VT API key
    VT_API: ...
    ## path to VT file scan reports
    VT_LOG: /var/log/bro/tmp/
  report: ${PYTHON36} virustotal.py
```

Android APK Detection powered by AndroPyTool

AndroPyTool is a tool for extracting static and dynamic features from Android APKs. It combines different well-known Android apps analysis tools such as DroidBox, FlowDroid, Strace, AndroGuard or VirusTotal analysis. Provided a source directory containing APK files, AndroPyTool applies all these tools to perform pre-static, static and dynamic analysis and generates files of features in JSON and CSV formats and also allows to save all the data in a MongoDB database.

AndroPyTool is configured for detection of APK files, whose MIME type is `application/vnd.android.package-archive` in IANA registry. The configuration is as below:

```
application:
  vnd.android.package-archive:
```

(continues on next page)

(continued from previous page)

```
    remote: true
    workdir: AndroPyTool
    environ:
      APK_LOG: /home/traffic/log/bro/tmp/
    install:
      - docker pull alexmyg/andropytool
    report: ${SHELL} detect.sh
```

Since the environment configuration of AndroPyTool is much too complex, we directly used its official Docker image for detection. Therefore, the AndroPyTool is called through *remote* detection mechanism, i.e. BroApt-Daemon server performs detection using AndroPyTool Docker image on APK files then send the report back to BroAPT-App framework for records.

Office Document Detection powered by MaliciousMacroBot

MaliciousMacroBot is to provide a powerful malicious file triage tool for cyber responders; help fill existing detection gaps for malicious office documents, which are still a very prevalent attack vector today; deliver a new avenue for threat intelligence, a way to group similar malicious office documents together to identify phishing campaigns and track use of specific malicious document templates.

MaliciousMacroBot is configured for detecting Office files, which is a document type based on XML, such as Microsoft Office and OpenOffice. The MIME types of such documents include `application/msword`, `application/ms-excel`, `application/vnd.ms-powerpoint` and `application/vnd.openxmlformats-officedocument.*`, etc. The configuration is as below:

```
application:
  vnd.openxmlformats-officedocument.*: &officedocument
    workdir: ${API_ROOT}/application/vnd.openxmlformats-officedocument/
    environ:
      MMB_LOG: /var/log/bro/tmp/
    install:
      - yum install -y git
      - git clone https://github.com/egaus/MaliciousMacroBot.git
      - ${PYTHON36} -m pip install ./MaliciousMacroBot/
      - yum clean -y all
    report: ${PYTHON36} MaliciousMacroBot-detect.py
    shared: officedocument
  msword: *officedocument
  vnd.ms-excel: *officedocument
  vnd.ms-powerpoint: *officedocument
  ...
```

Note: As you may have noticed here, the configured MIME types detected by MaliciousMacroBot has a * globing syntax, such shall be matched using *shell*-like globing mechanism.

As the MaliciousMacroBot detection method is shared by multiple MIME types, we set the `shared` key in the API to an identifier for the detection method, so that at runtime, such detection method will be *process-safe*.

Linux ELF Detection powered by ELF Parser

ELF Parser is designed for static ELF analysis. It can quickly determine the capabilities of an ELF binary through static analysis, then discover if the binary is known malware or a possible threat without ever executing the file.

ELF Parser is configured for the ELF file (MIME type: `application/x-executable` only. The configuration is as below:

```
application:
  x-executable:
    ## ELF Parser
    remote: true
    environ:
      ELF_LOG: /home/traffic/log/bro/tmp/
      ELF_SCORE: 100
    workdir: ELF-Parser
    install:
      - docker build --tag elfparser:1.4.0 --rm .
    report: ${SHELL} detect.sh
```

Common Linux Malware Detection powered by LMD

Linux Malware Detect (LMD) is a malware scanner for Linux, that is designed around the threats faced in shared hosted environments. It uses threat data from network edge intrusion detection systems to extract malware that is actively being used in attacks and generates signatures for detection. In addition, threat data is also derived from user submissions with the LMD checkout feature and from malware community resources. The signatures that LMD uses are MD5 file hashes and HEX pattern matches, they are also easily exported to any number of detection tools such as ClamAV.

LMD is configured for various common file types. The configuration is as below:

```
application:
  octet-stream: &lmd
    ## LMD
    workdir: ${API_ROOT}/application/octet-stream/LMD
    environ:
      LMD_LOG: /var/log/bro/tmp/
    install:
      - yum install -y git which
      - test -d ./linux-malware-detect/ ||
        git clone https://github.com/rfxn/linux-malware-detect.git
      - ${SHELL} install.sh
    report: ${SHELL} detect.sh
    shared: linux-maldet
text:
  html: *lmd
  x-c: *lmd
  x-perl: *lmd
  x-php: *lmd
```

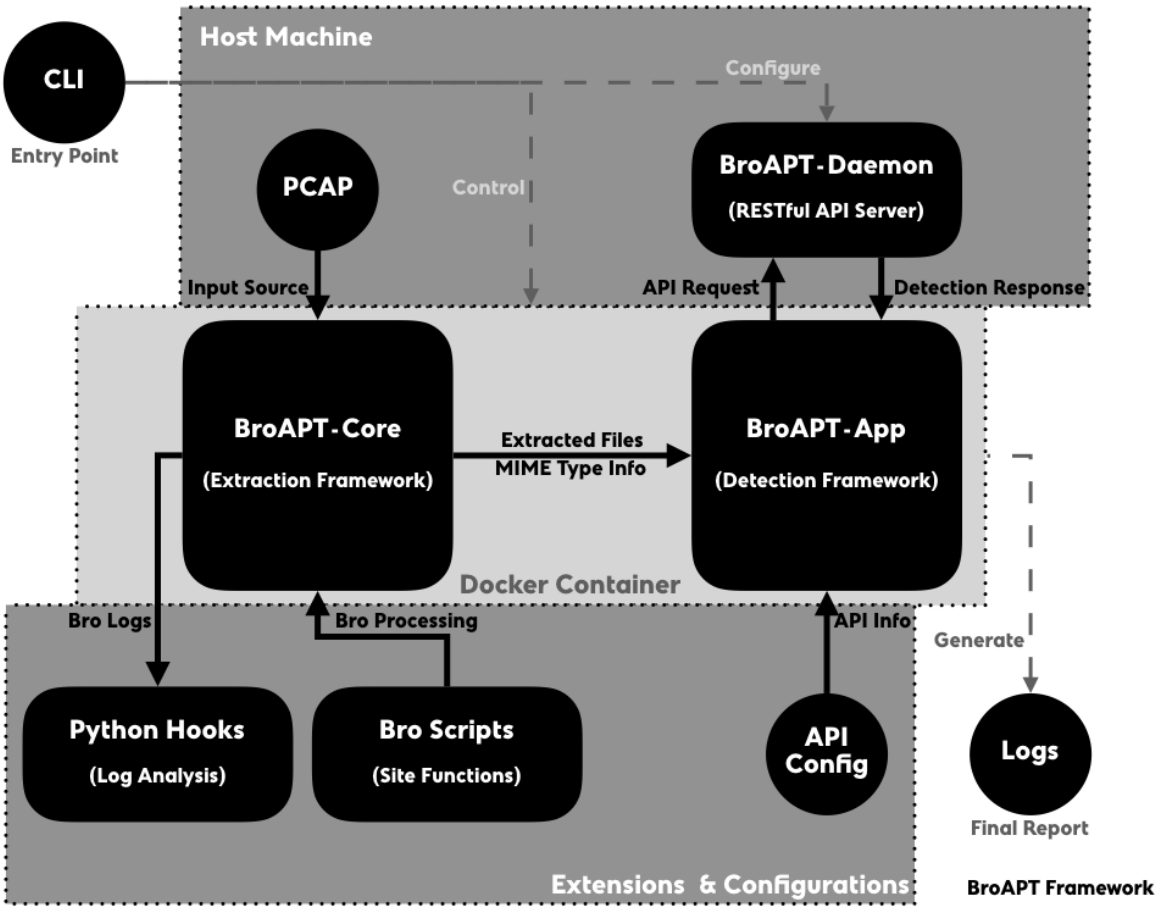
Malicious JavaScript Detection powered by JaSt

JaSt is a low-overhead solution that combines the extraction of features from the abstract syntax tree with a random forest classifier to detect malicious JavaScript instances. It is based on a frequency analysis of specific patterns, which are either predictive of benign or of malicious samples. Even though the analysis is entirely static, it yields a high detection accuracy of almost 99.5% and has a low false-negative rate of 0.54%.

JaSt as is dedicated for javaScript files. The configuration is as below:

```
application:
  javascript: &javascript
    workdir: ${API_ROOT}/application/javascript/JaSt
    environ:
      JS_LOG: /var/log/bro/tmp/
    install:
      - yum install -y epel-release
      - yum install -y git nodejs
      - test -d ./JaSt/ ||
        git clone https://github.com/Aurore54F/JaSt.git
      - ${PYTHON3} -m pip install
          matplotlib
          plotly
          numpy
          scipy
          scikit-learn
          pandas
      - ${PYTHON3} ./JaSt/clustering/learner.py
          --d ./sample/
          --l ./lables/
          --md ./models/
          --mn broapt-jast
    scripts:
    - ${PYTHON3} ./JaSt/clustering/classifier.py
        --f ${BROAPT_PATH}
        --m ./models/broapt-jast
    report: ${PYTHON3} detect.py
    shared: javascript
text:
  javascript: *javascript
```

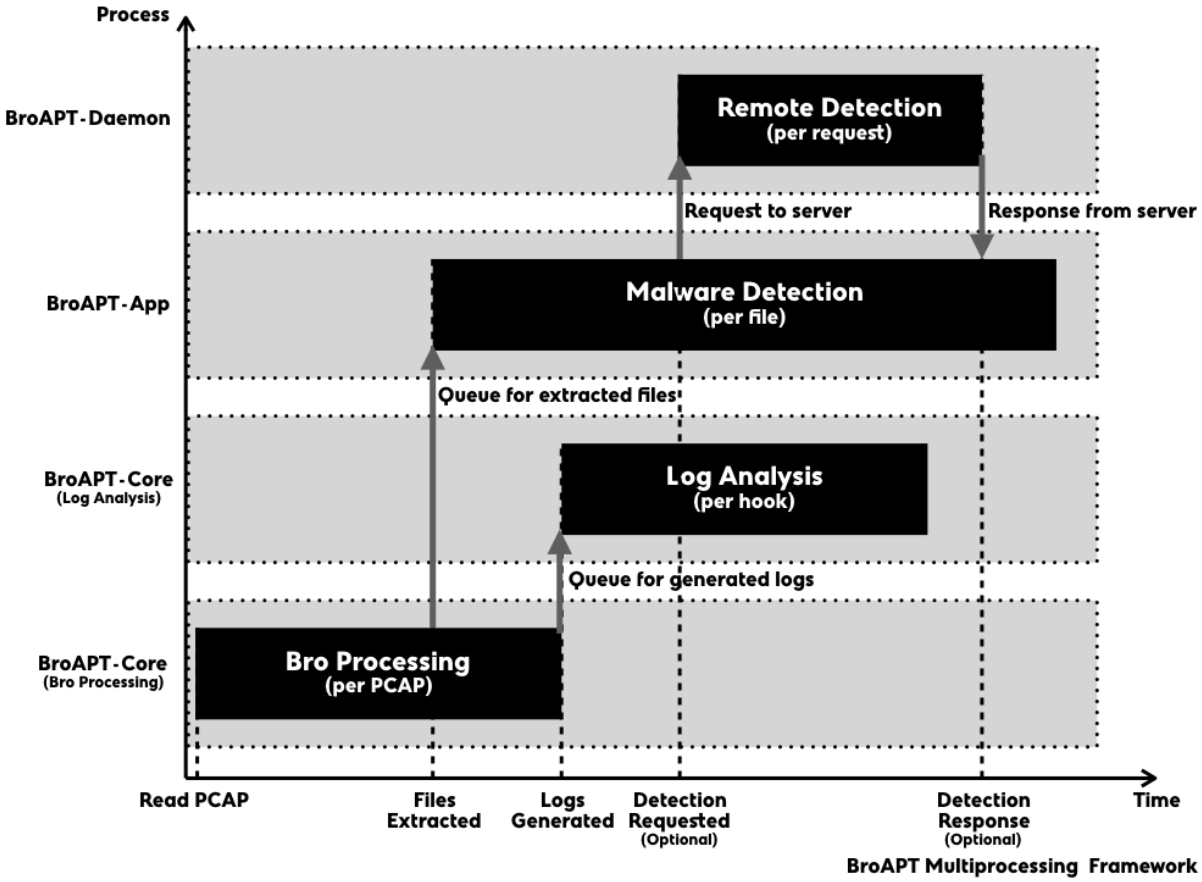
The BroAPT system is generally designed in two main parts, as we described in the *introduction*, the core functions and the daemon server with its command line interface (CLI).



On the host machine, the BroAPT-Daemon server runs as a manager of the BtoAPT system, which watches the running status of underlying BroAPT core functions, i.e. BroAPT-Core and BroAPT-App frameworks, as well as perform *remote* detection upon API requests from detection framework.

In the docker containers, the BroAPT-Core and BroAPT-App frameworks perform the core functions of BroAPT system. They analyse source PCAP files and extract files transferred through the traffic with Bro IDS, then detect the extracted files based on MIME type specifically configured APT detection methods.

The general process of processing is as following:



0. When the BroAPT-Core framework first reads a new PCAP file, it will utilise Bro IDS to process it, extract files transferred and perform other actions as configured through the Bro site functions.
1. As files had been extracted, the BroAPT-App framework will perform malware detection on each file. If *remote* detection configured, it will send an API request to the BroAPT-Daemon server, and wait for its detection report.
2. At the same time, once the Bro processing had finished, the BroAPT-Core framework will start processing the generated logs, and perform extra analysis over the Bro log files as specified by the Python hooks.
3. When the BroAPT-Daemon receives an API request, it will perform malware detection as described in the request, and send the detection report back to the BroAPT-App framework.

3.3 Implementation Details

In first draft design, the BroAPT system was implemented in a cluster manner, comparing to current bundled distribution, i.e. the BroAPT-Core framework and BroAPT-App framework are two separate Docker containers. However, two implementation manners are both maintained at the moment.

Note: In the documentation, we normally refer to bundled implementation when talking about the BroAPT system internal implementation details.

Through internal module name may vary between two implementations, the main implementation source codes is, nevertheless, identical in both implementations.

3.3.1 Cluster Implementation

Note: For source codes, please go to `/source/` folder.

In the cluster implementation, the BroAPT-Core framework is running in a CentOS 7 container, as the then-latest version of Bro IDS (version 2.6.1) was only available through RPM binary; whilst the BroAPT-App framework is running is in an Ubuntu 16.04 container, with better compatibility for detection tools.

The communication between two frameworks is archived through file system temporary listing files.

3.3.2 Bundled Implementation

Note: For source codes, please go to `/cluster/` folder.

In the bundled implementation, both the BroAPT-Core framework and the BroAPT-App framework are running in a CentOS 7 container.

The communication between two frameworks is archived through `multiprocessing.Queue`.

CHAPTER

FOUR

API REFERENCE

As discussed previously, the BroAPT system has two different implementation architectures. They are similar in overall concepts and processing, but may various in underlying internal source codes. We'll try to break down into details of each implementation for you to develop new extensions, hooks, scripts for the BroAPT system in *humans* way.

4.1 BroAPT-Core Framework

The BroAPT-Core framework is the extraction framework for the BroAPT system. For more information about the framework, please refer to previous documentation at *BroAPT-Core Extration Framework*.

4.1.1 Bro Scripts

Module Entry

File location

- Bundled implementation: `source/client/scripts/__load__.bro`
- Cluster implementation: `cluster/core/source/scripts/__load__.bro`

This is the entry point of the Bro scripts.

Configurations

File location

- Bundled implementation: `source/client/scripts/config.bro`
- Cluster implementation: `cluster/core/source/scripts/config.bro`

This file contains custom configurations for the Bro IDS at runtime. It will be automatically regenerated at runtime through the **Bro script composer**, based on the following environment variables:

- *BROAPT_LOGS_PATH*
- *BROAPT_PCAP_PATH*
- *BROAPT_MIME_MODE*
- *BROAPT_HASH_MD5*
- *BROAPT_HASH_SHA1*
- *BROAPT_HASH_SHA256*

- *BROAPT_X509_MODE*
- *BROAPT_ENTROPY_MODE*
- *BROAPT_DUMP_PATH*
- *BROAPT_FILE_BUFFER*
- *BROAPT_SIZE_LIMIT*
- *BROAPT_JSON_MODE*
- *BROAPT_LOAD_MIME*
- *BROAPT_LOAD_PROTOCOL*

MIME-Extension Mappings

File location

- Bundled implementation: `source/client/scripts/file-extensions.bro`
- Cluster implementation: `cluster/core/source/scripts/file-extensions.bro`

This file contains a Bro `table` mapping MIME types to possible file extensions. The MIME types are fetched from IANA registries and the file extensions are provided *semi-automatically* through `mimetypes` database.

This Bro script can be generated from the the `mime2ext.py` script as we described in the *Miscellaneous & Auxiliary* section.

`FileExtraction` Module

File location

- Bundled implementation: `source/client/scripts/main.bro`
- Cluster implementation: `cluster/core/source/scripts/main.bro`

This files is the main implementation of the `FileExtraction` module. The main logic can be simplified as following Bro script:

```
module FileExtraction;

event file_sniff(f: fa_file, meta: fa_metadata) {
    if ( !hook FileExtraction::ignore(f, meta) )
        return;

    if ( !hook FileExtraction::extract(f, meta) ) {
        # scripts to generate an output file name
        local name: string = ...;

        # extract the file to the ``name``
        Files::add_analyzer(f, Files::ANALYZER_EXTRACT, [$extract_filename=name]);
    }
}
```

where `FileExtraction::ignore` and `FileExtraction::extract` are the two Bro `hook` functions, i.e. predicates, you may customise to affect the extraction behaviour.

Extract by Protocol

File location

- Bundled implementation: `source/client/scripts/hooks/`
- Cluster implementation: `cluster/core/source/scripts/hooks/`

This fold contains Bro `hook` functions to toggle if extract files transferred through a certain application layer protocol. Such scripts will be loaded based on *BROAPT_LOAD_PROTOCOL* environment variable.

Supported protocols are:

- DTLS
- FTP
- HTTP
- IRC
- SMTP

To extract all files transferred through HTTP, i.e. `extract-http.bro` in the folder, the Bro `hook` function should be as below:

```
@load ../__load__.bro
@load base/protocols/http/entities.bro

module FileExtraction;

hook FileExtraction::extract(f: fa_file, meta: fa_metadata) &priority=15 {
   if ( f$source == "HTTP" )
      break;
}
```

Note: We load `base/protocols/http/entities.bro` to support the script even running in *bare* mode.

Extract by MIME Type

File location

- Bundled implementation: `source/client/scripts/plugins/`
- Cluster implementation: `cluster/core/source/scripts/plugins/`

This fold contains Bro `hook` functions to toggle if extract files of a certain MIME type. Such files will be generated based on *BROAPT_LOAD_MIME* environment variable.

To extract all files, i.e. `extract-all-files.bro` in the folder, the Bro `hook` function should be as below:

```
@load ../__load__.bro

module FileExtraction;

hook FileExtraction::extract(f: fa_file, meta: fa_metadata) &priority=10 {
   break;
}
```

Site Customisations

File location

- Bundled implementation: `source/include/scripts/`
- Cluster implementation: `cluster/core/include/scripts/`

This folder will be mapped into the Docker container as `/broapt/scripts/sites/`. You may load your customised script in the `__load__.bro` file.

Note: Should the `sites` folder doesn't exist, it will not be loaded into the main scripts to avoid raising errors at runtime.

Currently, we have integrated six sets of customised Bro scripts, please see *BroAPT-Core Extration Framework* for more information.

4.1.2 Python Modules

Module Entry

File location

- Bundled implementation: `source/client/python/__init__.py`
- Cluster implementation: `cluster/core/source/python/__init__.py`

This file merely modifies the `sys.path` so that we can import the Python modules as if from the top level.

System Entrypoint

File location

- Bundled implementation: `source/client/python/__main__.py`
- Cluster implementation: `cluster/core/source/python/__main__.py`

This file wraps the whole system and make the `python` folder callable as a module where the `__main__.py` will be considered as the entrypoint.

`__main__.PCAP_MGC = (b'\xa1\xb2\x3c\x4d', b'\xa1\xb2\xc3\xd4', b'\x4d\x3c\xb2\xa1', b'\xd4\xc3\xb2\xa1', b'\x0a\x0d\x0d\x0a')`

A tuple of magic numbers for PCAP files:

```
a1 b2 3c 4d  # PCAP files in big endian with nanosecond timestamp
a1 b2 c3 d4  # PCAP files in big endian
4d 3c b2 a1  # PCAP files in little endian with nanosecond timestamp
d4 c3 b2 a1  # PCAP files in little endian
0a 0d 0d 0a  # PCAPng files
```

`__main__.is_pcap`(*file: str*)

Check if `file` is a valid PCAP file with help of `libmagic`.

Parameters

`file` (*str*) – Path of the file to be checked.

Returns
If is a valid PCAP file.

Return type
bool

__main__.**listdir**(*path: str*)

Fetch all files under `path`.

Parameters
path (*str*) – Path to be fetched.

Return type
List[str]

__main__.**parse_args**(*argv: List[str]*)

Parse command line arguments (path to PCAP files) and fetch valid PCAP files.

Note: If a directory is provided, it will be recursively listed with *listdir()*.

Parameters
argv (*List[str]*) – Command line arguments.

Returns
List of valid PCAP files.

Return type
List[str]

__main__.**check_history**()

Check processed PCAP files.

Note: Processed PCAP files will be recorded at *const.FILE*.

Returns
List of processed PCAP files.

Return type
List[str]

__main__.**main_with_args**()

Run the BroAPT system **with** command line arguments.

Note: The process will exit once all PCAP files fetched from the paths given by the command line arguments are processed.

Returns
Exit code.

Return type
int

__main__.**main_with_no_args**()

Run the BroAPT system **without** command line arguments.

Note: The process will run and check for new PCAP files from *const.PCAP_PATH* indefinitely.

__main__.**main**()

Run the BroAPT-App framework under the context of *remote.remote_proc()*.

Returns
Exit code.

Return type
int

See also:

- *main_with_args()*
- *main_with_no_args()*

Bro Script Composer

File location

- Bundled implementation: `source/client/python/compose.py`
- Cluster implementation: `cluster/core/source/python/compose.py`

Note: This file works as a standalone script for generating Bro scripts. It is **NOT** meant to be an *importable* module of the BroAPT system.

Introduction

As we can config what MIME types to extract through the *BROAPT_LOAD_MIME* environment variable, the BroAPT-Core framework will automatically generate the Bro scripts based on this environment variable and many others.

For MIME types with a *shell*-like pattern, we will use `fnmatch.translate()` to convert the pattern into a regular expression.

A generated *Bro script* for `hook` function extracting files with MIME type `example/test-*` would be as following:

```
@load ../__load__.bro

module FileExtraction;

hook FileExtraction::extract(f: fa_file, meta: fa_metadata) &priority=5 {
    if ( meta?$mime_type && /example\/test\-.*/ == meta$mime_type )
        break;
}
```

Besides this, the Bro script composer will also generate/rewrite the *Bro configurations* to customise several metrics and to load the scripts as specified in the environment variables.

Note: The full list of supported environment variables is as following:

- *BROAPT_LOGS_PATH*
- *BROAPT_PCAP_PATH*
- *BROAPT_MIME_MODE*
- *BROAPT_HASH_MD5*
- *BROAPT_HASH_SHA1*
- *BROAPT_HASH_SHA256*
- *BROAPT_X509_MODE*
- *BROAPT_ENTROPY_MODE*
- *BROAPT_DUMP_PATH*
- *BROAPT_FILE_BUFFER*
- *BROAPT_SIZE_LIMIT*
- *BROAPT_JSON_MODE*
- *BROAPT_LOAD_MIME*
- *BROAPT_LOAD_PROTOCOL*

Functions

compose.**file_salt**(*uid: str*)

Update the `config.bro` (*Configurations*) with provided `uid` as `file_salt`.

compose.**compose**()

Compose Bro scripts with environment variables defined.

Note: This function is the module entry.

compose.**escape**(*mime_type: str*)

Escape *shell*-like `mime_type` pattern to regular expression.

Caution: The underlying implementation of `fnmatch.translate()` calls `re.escape()` to escape special characters. However, in Python 3.6, the function will escape all characters other than ASCIIs, numbers and underlines (_); whilst in Python 3.7, it will only escape characters defined in `re._special_chars_map`.

Constants

Auxiliaries

compose.**ROOT**

Type
str

Path to the BroAPT-Core framework source codes (absolute path at runtime).

compose.**BOOLEAN_STATES = {'1': True, '0': False, 'yes': True, 'no': False, 'true': True, 'false': False, 'on': True, 'off': False}**

Mapping of boolean states, c.f. configparser.

Bro Configs

compose.**LOGS_PATH**

Type
str (path)

Environ
BROAPT_LOGS_PATH

Path to system logs.

compose.**PCAP_PATH**

Type
str (path)

Environ
BROAPT_PCAP_PATH

Path to source PCAP files.

compose.**MIME_MODE**

Type
bool

Environ
BROAPT_MIME_MODE

If group extracted files by MIME type.

compose.**HASH_MODE_MD5**

Type
bool

Environ
BROAPT_HASH_MD5

Calculate MD5 hash of extracted files.

compose.**HASH_MODE_SHA1**

Type
bool

Environ
BROAPT_HASH_SHA1

Calculate SHA1 hash of extracted files.

compose.**HASH_MODE_SHA256**

Type
bool

Environ
BROAPT_HASH_SHA256

Calculate SHA256 hash of extracted files.

compose.**X509_MODE**

Type
bool

Environ
BROAPT_X509_MODE

Include X509 information when running Bro.

compose.**ENTROPY_MODE**

Type
bool

Environ
BROAPT_ENTROPY_MODE

Include file entropy information when running Bro.

compose.**DUMP_PATH**

Type
str (path)

Environ
BROAPT_DUMP_PATH

Path to extracted files.

. data:: compose.FILE_BUFFER

type
int (uint64)

environ
BROAPT_FILE_BUFFER

Reassembly buffer size for file extraction.

compose.**SIZE_LIMIT**

Type
int (uint64)

Environ
BROAPT_SIZE_LIMIT

Size limit of extracted files.

compose.**JSON_MODE**

Type
bool

Environ
BROAPT_JSON_MODE

Toggle Bro logs in JSON or ASCII format.

compose.**LOAD_MIME**

Type
List[str] (*case-insensitive*)

Environ
BROAPT_LOAD_MIME

A `,` or `;` separated string of MIME types to be extracted.

compose.**LOAD_PROTOCOL**

Type
List[str] (*case-insensitive*)

Environ
BROAPT_LOAD_PROTOCOL

A `,` or `;` separated string of application layer protocols to be extracted, can be any of `dtls`, `ftp`, `http`, `irc` and `smtp`.

Subsitute Patterns

compose.**FILE_TEMP**

Type
Tuple[str]

Template for MIME type extraction Bro scripts.

compose.**MIME_REGEX**

Type
re.Pattern

Pattern for `mime` (*MIME_MODE*).

compose.**LOGS_REGEX**

Type
re.Pattern

Pattern for `logs` (*LOGS_PATH*).

compose.**HASH_REGEX_MD5**

Type
re.Pattern

Pattern for `md5` (*HASH_MODE_MD5*).

compose.**HASH_REGEX_SHA1**

Type
re.Pattern

Pattern for `sha1` (*HASH_MODE_SHA1*).

compose.**HASH_REGEX_SHA256**

Type
re.Pattern

Pattern for `sha256` (*HASH_MODE_SHA256*).

compose.**X509_REGEX**

Type
re.Pattern

Pattern for `x509` (*X509_MODE*).

compose.**ENTR_REGEX**

Type
re.Pattern

Pattern for `entropy` (*ENTROPY_MODE*).

compose.**JSON_REGEX**

Type
re.Pattern

Pattern for `use_json` (*JSON_MODE*).

compose.**SALT_REGEX**

Type
re.Pattern

Pattern for `file_salt` (`file_salt()`).

compose.**FILE_REGEX**

Type
re.Pattern

Pattern for `file_buffer` (`FILE_BUFFER`).

compose.**PATH_REGEX**

Type
re.Pattern

Pattern for `path_prefix` (*DUMP_PATH*).

compose.**SIZE_REGEX**

Type
re.Pattern

Pattern for `size_limit` (*SIZE_LIMIT*).

compose.**LOAD_REGEX**

Type
re.Pattern

Pattern for `@load` loading scripts.

Common Constants

File location

- Bundled implementation: `source/client/python/const.py`
- Cluster implementation: `cluster/core/source/python/const.py`

const.**ROOT**

Type
str

Path to the BroAPT-Core framework source codes (absolute path at runtime).

const.**BOOLEAN_STATES = {'1': True, '0': False, 'yes': True, 'no': False, 'true': True, 'false': False, 'on': True, 'off': False}**

Mapping of boolean states, c.f. `configparser`.

const.**CPU_CNT**

Type
int

Environ
BROAPT_CPU

Number of BroAPT concurrent processes for PCAP analysis. If not provided, then the number of system CPUs will be used.

const.**INTERVAL**

Type
float

Environ

- Bundled implementation: *BROAPT_INTERVAL*
- Cluster implementation: *BROAPT_CORE_INTERVAL*

Wait interval after processing current pool of PCAP files.

const.**DUMP_PATH**

Type
str (path)

Environ
BROAPT_DUMP_PATH

Path to extracted files.

const.**PCAP_PATH**

Type
`str` (path)

Environ
BROAPT_PCAP_PATH

Path to source PCAP files.

const.**LOGS_PATH**

Type
`str` (path)

Environ
BROAPT_LOGS_PATH

Path to system logs.

const.**MIME_MODE**

Type
`bool`

Environ
BROAPT_MIME_MODE

If group extracted files by MIME type.

const.**BARE_MODE**

Type
`bool`

Environ
BROAPT_BARE_MODE

Run Bro in bare mode (don't load scripts from the `base/` directory).

const.**NO_CHKSUM**

Type
`bool`

Environ
BROAPT_NO_CHKSUM

Ignore checksums of packets in PCAP files when running Bro.

const.**HOOK_CPU**

Type
`int`

Environ
BROAPT_HOOK_CPU

Number of BroAPT concurrent processes for Python hooks.

const.**FILE**

Type
str

```
os.path.join(LOGS_PATH, 'file.log')
```

Path to file system database of processed PCAP files.

const.**TIME**

Type
str

```
os.path.join(LOGS_PATH, 'time.log')
```

Path to log file of processing time records.

const.**STDOUT**

Type
str

```
os.path.join(LOGS_PATH, 'stdout.log')
```

Path to `stdout` *replica*.

const.**STDERR**

Type
str

```
os.path.join(LOGS_PATH, 'stderr.log')
```

Path to `stderr` *replica*.

const.**QUEUE_LOGS**

Type
`multiprocessing.Queue`

Availability
bundled implementation

Teleprocess communication queue for log processing.

const.**QUEUE**

Type
`multiprocessing.Queue`

Availability
cluster implementation

See also:

const.QUEUE_LOGS

Bro Log Parser

File location

- Bundled implementation: `source/client/python/logparser.py`
- Cluster implementation: `cluster/core/source/python/logparser.py`

Important: This module has been deprecated for production reasons. Please use the ZLogging module for parsing Bro logs.

Dataclasses

class logparser.**TEXTInfo**

A dataclass for parsed ASCII log file.

format = 'text'

Log file format.

path: *str*

Path to log file.

open: *datetime.datetime*

Open time of log file.

close: *datetime.datetime*

Close time of log file.

context: *pandas.DataFrame*

Parsed log context.

exit_with_error: *bool*

If log file exited with error, i.e. close time *close* doesn't present in the log file.

class logparser.**JSONInfo**

A dataclass for parsed JSON log file.

format = 'json'

Log file format.

context: *pandas.DataFrame*

Parsed log context.

Field Parsers

logparser.**set_separator:** *str*

Separator of `set` & `vector` values in ASCII logs.

logparser.**empty_field:** *str*

Separator of *empty* fields in ASCII logs.

logparser.**unset_field**: *str*

Separator of *unset* fields in ASCII logs.

Note: If the field is `unset_field`, then the parsers below will return `None`.

logparser.**set_parser**(*s: str*, *t: Type[T]*)

Parse `set` field.

Parameters

- **s** (*str*) – Field string.
- **t** (*type*) – Type of `set` elements.

Return type
`Set[T]`

logparser.**vector_parser**(*s: str*, *t: Type[T]*)

Parse `vector` field.

Parameters

- **s** (*str*) – Field string.
- **t** (*type*) – Type of `vector` elements.

Return type
`List[T]`

logparser.**str_parser**(*s: str*)

Parse `string` field.

Parameters
s (*str*) – Field string.

Return type
`str`

Note: To *unescape* the escaped bytes characters, we use the `unicode_escape` encoding to decode the parsed string.

logparser.**port_parser**(*s: str*)

Parse `port` field.

Parameters
s (*str*) – Field string.

Return type
`int` (`uint16`)

logparser.**int_parser**(*s: str*)

Parse `int` field.

Parameters
s (*str*) – Field string.

Return type
`int` (`int64`)

logparser.**count_parser**(*s: str*)

Parse count field.

Parameters
s (*str*) – Field string.

Return type
int (uint64)

logparser.**addr_parser**(*s: str*)

Parse addr field.

Parameters
s (*str*) – Field string.

Return type
Union[ipaddress.IPv4Address, ipaddress.IPv6Address]

logparser.**subnet_parser**(*s: str*)

Parse subnet field.

Parameters
s (*str*) – Field string.

Return type
Union[ipaddress.IPv4Network, ipaddress.IPv6Network]

logparser.**time_parser**(*s: str*)

Parse time field.

Parameters
s (*str*) – Field string.

Return type
datetime.datetime

logparser.**float_parser**(*s: str*)

Parse float field.

Parameters
s (*str*) – Field string.

Return type
decimal.Decimal (*precision* set to 6)

logparser.**interval_parser**(*s: str*)

Parse interval field.

Parameters
s (*str*) – Field string.

Return type
datetime.timedelta

logparser.**enum_parser**(*s: str*)

Parse enum field.

Parameters
s (*str*) – Field string.

Return type
enum.Enum

logparser.**bool_parser**(*s: str*)

Parse `bool` field.

Parameters
s (*str*) – Field string.

Return type
`bool`

Raises
ValueError – If `s` is not a valid value, i.e. any of *unset_field*, `'T'` (`True`) or `'F'` (`False`).

logparser.**type_parser = collections.defaultdict(lambda: str_parser, dict(string=str_parser, port=port_parser, enum=enum_parser, interval=interval_parser, addr=addr_parser, subnet=subnet_parser, int=int_parser, count=count_parser, time=time_parser, double=float_parser, bool=bool_parser,))**

Mapping for Bro types and corresponding parser function.

Log Parsers

logparser.**parse_text**(*file: io.TextIOWrapper, line: str, hook: Optional[Dict[str, Callable[[str], Any]]]*)

Parse ASCII logs.

Parameters

- **file** – Log file opened in read (`'r'`) mode.
- **line** (*str*) – First line of the log file (used for format detection by *parse()*).
- **hook** – Addition parser mappings to register in *type_parser*.

Return type
TEXTInfo

logparser.**parse_text**(*file: io.TextIOWrapper, line: str*)

Parse JSON logs.

Parameters

- **file** – Log file opened in read (`'r'`) mode.
- **line** (*str*) – First line of the log file (used for format detection by *parse()*).

Return type
JSONInfo

logparser.**parse**(*filename: str, hook: Optional[Dict[str, Callable[[str], Any]]]*)

Parse Bro logs.

Parameters

- **filename** (*str*) – Log file to be parsed.
- **hook** – Addition parser mappings to register in *type_parser* when processing ASCII logs for *parse_text()*.

Return type
Union[*TEXTInfo*, *JSONInfo*]

Note: The function will automatically detect if the given log file is in ASCII or JSON format.

Module Entry

logparser.**main**()

```
python logparser.py [filename ...]
```

Wrapper function to parse and *pretty* print log files.

Extraction Process

File location

- Bundled implementation: `source/client/python/process.py`
- Cluster implementation: `cluster/core/source/python/process.py`

process.**process**(*file: str*)

Process PCAP file with Bro IDS and put the root folder to Bro logs into *const.QUEUE_LOGS*.

Parameters
file (*str*) – Path to PCAP file.

communicate(*log_root: str*)

Check if extracted files exist based on `extracted` field from the `files.log`.

In **bundled implementation**, then put the files into *const.QUEUE_DUMP*.

Parameters
log_root (*str*) – Root folder to Bro logs.

Raises
ExtractWarning – When supposedly extracted file not found.

process.**SALT_LOCK: multiprocessing.Lock**

Lock for updating `config.bro` with `compsoe.file_salt()`.

process.**STDOUT_LOCK: multiprocessing.Lock**

Lock for writing to the `stdout` *replica* *const.STDOUT*.

process.**STDERR_LOCK: multiprocessing.Lock**

Lock for writing to the `stderr` *replica* *const.STDERR*.

exception process.**ExtractWarning**

Bases
Warning

Extraction warning.

Bro Logs Processing

File location

- Bundled implementation: `source/client/python/remote.py`
- Cluster implementation: `cluster/core/source/python/remote.py`

Hook Mainloop

remote.**remote_proc**()

A context for running processes at the background.

In **bundled implementation**, this function also starts both *remote_dump()* and *remote_logs()* as new processes.

In **cluster implementation**, this function starts *remote()* as a new process.

Note: Before exit, in **bundled implementation**, it will send `SIGUSR1` signal to the *remote_dump()* background process and `SIGUSR2` signal to the *remote_logs()* background process; then wait for the process to gracefully exit.

In **cluster implementation**, it will send `SIGUSR1` signal to the *remote_logs()* background process and wait for the process to gracefully exit.

---

remote.**remote_logs**()

Availability
bundled implementation

Runtime mainloop for Python hooks.

The function will start as an *indefinite* loop to fetch path to Bro logs from *const.QUEUE_LOGS*, and execute registered Python hooks on them.

When *JOIN_LOGS* is set to `True`, the function will break from the loop and execute registered Python hooks for *closing* (*sites.EXIT*).

Raises
HookWarning – If hook execution failed.

remote.**remote**()

Availability
cluster implementation

The function will start as an *indefinite* loop to fetch path to Bro logs from *const.QUEUE*, and execute registered Python hooks on them.

When *JOIN* is set to `True`, the function will break from the loop and execute registered Python hooks for *closing* (*sites.EXIT*).

Raises
HookWarning – If hook execution failed.

hook(*log_name: str*)

Wrapper function for running registered Python hooks.

Parameters
log_name (*str*) – Root folder of Bro logs.

wrapper_logs(*args: Tuple[Callable[[str], Any], str]*)

Wrapper function for running registered Python hooks for *processing* (*sites.HOOK*).

wrapper_func(*func: Callable[[], Any]*)

Wrapper function for running registered Python hooks for *closing* (*sites.EXIT*).

Warnings

exception remote.**HookWarning**

Bases
Warning

Warns when Python hooks execution failed.

Signal Handling

Bundled Implementation

remote.**join_logs**(**args*, ***kwargs*)

Availability
bundled implementation

Toggle *JOIN_LOGS* to True.

Note: This function is registered as handler for SIGUSR2`.

remote.**JOIN_LOGS = multiprocessing.Value('B', False)**

Availability
bundled implementation

Flag to stop the *remote_logs()* background process.

Cluster Implementation

remote.**join**(**args*, ***kwargs*)

Availability
cluster implementation

Toggle *JOIN* to True.

Note: This function is registered as handler for SIGUSR1`.

remote.**JOIN = multiprocessing.Value('B', False)**

Availability
cluster implementation

Flag to stop the *remote()* background process.

Auxiliaries & Utilities

File location

- Bundled implementation: `source/client/python/utils.py`
- Cluster implementation: `cluster/core/source/python/utils.py`

`@utils.`**`suppress`**

A decorator that suppresses all exceptions.

`utils.`**`file_lock`**(*file: str*)

A context lock for file modification with a file system lock.

Parameters
`file` (*str*) – Filename to be locked in the context.

`utils.`**`print_file`**(*s: Any*, *file: str*)

Wrapper function to *process*-safely print `s` into `file`.

Parameters

- **`s`** (*str*) – Content to be printed.
- **`file`** (*str*) – Filename of output stream.

`utils.`**`redirect`**(*src: str*, *dst: str*, *label='unknown'*)

Redirect the content of `src` to `dst` with `label` as prefix:

```
<label> line from src
```

Parameters

- **`src`** (*str*) – Filename of source file.
- **`dst`** (*str*) – Filename of destination file.
- **`label`** (*str*) – Optional prefix to the redirected content.

`utils.`**`is_nan`**(*value: Any*)

Check if `value` is `None` or a `NaN`.

Parameters
`value` – Value to be checked.

Return type
bool

Site Customisations

File location

- Bundled implementation: `source/include/python/`
- Cluster implementation: `cluster/core/include/python/`

This folder will be mapped into the Docker container as `/broapt/python/sites/`. You may register your customised Python hooks in the `__init__.py` file.

sites.**HOOK: List[Callable[[str], Any]]**

Registry for *processing* hooks.

Registered function should take the path to the folder of Bro logs as a single parameter, return values will be ignored. Such functions will be called on each Bro log folder generated from PCAP files.

sites.**EXIT: List[Callable[[], Any]]**

Registry for *closing* hooks.

Registered function should take **NO** parameters, return values will be ignored. Such functions will be called before the system exits.

Currently, we have integrated two sets of customised Python hooks, please see *BroAPT-Core Extration Framework* for more information.

4.1.3 Wrapper Scripts

For the Docker container, we have created some Shell/Bash wrapper scripts to make the life a little bit better.

Bundled Implementation

File location

source/client/init.sh

```
#!/usr/bin/env bash

set -aex

# change curdir
cd /broapt

# load environs
if [ -f .env ] ; then
    source .env
fi

# compose Bro scripts
/usr/bin/python3.6 python/compose.py

# run scripts
/usr/bin/python3.6 python $@

# sleep
sleep infinity
```

Cluster Implementation

File location
cluster/core/source/init.sh

```
#!/usr/bin/env bash

set -aex

# change cwd
cd /source

# load environs
if [ -f .env ] ; then
    source .env
fi

# compose Bro scripts
/usr/bin/python3.6 python/compose.py

# run scripts
/usr/bin/python3.6 python $@

# sleep
sleep infinity
```

4.2 BroAPT-App Framework

The BroAPT-App framework is the analysis framework for the BroAPT system. For more information about the framework, please refer to previous documentation at *BroAPT-App Detection Framework*.

4.2.1 Python Modules

Module Entry

File location

- Bundled implementation: source/client/python/__init__.py
- Cluster implementation: cluster/app/source/python/__init__.py

This file merely modifies the sys.path so that we can import the Python modules as if from the top level.

System Entrypoint

File location

- Bundled implementation:
  - `source/client/python/remote.py`
  - `source/client/python/scan.py`
- Cluster implementation: `cluster/app/source/python/__main__.py`

In **bundled implementation**, the *Bro Logs Processing* module (`remote`) starts a background process for the BroAPT-App framework; whilst the *Detection Process* module (`process`) contains *main processing logic* as well as the original system entrypoint.

In **cluster implementation**, this file wraps the whole system and make the `python` folder callable as a module where the `__main__.py` will be considered as the entrypoint.

Constants

`__main__.FILE_REGEX: re.Pattern`

Availability
cluster implementation

```
re.compile(r'''
    # protocol prefix
    (?P<protocol>DTLS|FTP_DATA|HTTP|IRC_DATA|SMTP|\S+)
    -
    # file UID
    (?P<fuid>F\w+)
    \.
    # PCAP source
    (?P<pcap>.+?)
    \.
    # media-type
    (?P<media_type>
↪application|audio|example|font|image|message|model|multipart|text|video|\S+)
    \.
    # subtype
    (?P<subtype>\S+)
    \.
    # file extension
    (?P<extension>\S+)
''', re.IGNORECASE | re.VERBOSE)
```

Regular expression to match and fetch information from extracted files.

See also:

const.FILE_REGEX

Dataclasses

class scan.**MIME**

Availability
bundled implementation

A dataclass for parsed MIME type.

media_type: *str*
Media type.

subtype: *str*
Subtype.

name: *str*
MIME type.

class __main__.**MIME**

Availability
cluster implementation

See also:

scan.MIME

class scan.**Entry**

Availability
bundled implementation

A dataclass for extracted file entry.

path: *str*
File path.

uuid: *str*
UUID parsed from file.

mime: *MIME*
Parsed MIME type *dataclass*.

Note: This dataclass supports ordering with power of `functools.total_ordering()`.

---

class __main__.**Entry**

Availability
cluster implementation

See also:

scan.Entry

Bundled Implementation

scan Module

scan.**scan**(*local_name: str*)

Availability
bundled implementation

Parse then start processing of the given file.

See also:

scan.process()

scan.**lookup**(*path: str*)

Availability
bundled implementation

Fetch all extracted files to be processed from the given path.

Parameters
path (*str*) – Path to extracted files.

Returns
List of extracted files.

Return type
List[str]

remote Module

Framework Mainloop

remote.**remote_dump**()

Availability
bundled implementation

Runtime mainloop for BroAPT-App framework.

The function will start as an *indefinite* loop to fetch path to extracted files from *const.QUEUE_DUMP*, and perform *scan()* on them.

When *JOIN_DUMP* is set to True, the function will break from the loop.

Signal Handling

remote.**join_dump**(**args*, ***kwargs*)

Availability
bundled implementation

Toggle *JOIN_DUMP* to True.

Note: This function is registered as handler for SIGUSR1`.

remote.**JOIN_DUMP = multiprocessing.Value('B', False)**

Availability
bundled implementation

Flag to stop the *remote_dump()* background process.

Cluster Implementation

__main__.**listdir**(*path: str*)

Availability
cluster implementation

Fetch and parse all extracted files in the given path.

Parameters
path (*str*) – Path to extracted files.

Returns
List of parsed *entry* for extracted files.

Return type
List[*Entry*]

__main__.**check_history**()

Availability
cluster implementation

Check processed extracted files.

Note: Processed extracted files will be recorded at *const.DUMP*.

Returns
List of processed extracted files.

Return type
List[str]

__main__.**main**()

Availability
cluster implementation

Run the BroAPT-Core framework.

Returns
Exit code.

Return type
int

See also:

__main__.process()

API Config Parser

File location

- Bundled implementation: `source/client/python/cfgparser.py`
- Cluster implementation: `cluster/app/source/python/cfgparser.py`

Dataclasses

class cfgparser.**API**

A dataclass for parsed API entry.

workdir: str

API working directory.

environ: Dict[str, Any]

API runtime environment variables.

install: List[str | List[str]]

List of installation commands.

scripts: List[str | List[str]]

List of detection commands.

report: str

Report generation command.

remote: bool

If the API required *remote* execution, i.e. through the BroAPT-Daemon server.

shared: str

Sharing identifier, i.e. which MIME type the API entry is shared with.

inited = multiprocessing.Value('B', False)

Initied flag.

locked: multiprocessing.Lock

Multiprocessing runtime lock.

Functions

cfgparser.**parse_cmd**(*context: Dict[str, Any]*, *mimetype: str*, *environ: Dict[str, Any]*)

Parse API of `mimetype`.

Parameters

- **context** – API configuration context.
- **mimetype** (*str*) – MIME type of the API.
- **environ** – Global environment variables.

Raises

ReportNotFoundError – If `report` section not presented in `context`.

cfgparser.**parse**(*root: str*)

Parse API configuration file `api.yml`.

Parameters
root (*str*) – Root path to the APIs.

Returns
The parsed API entries, i.e. *API_DICT*.

Return type
Dict[str, *API*]

Constants

cfgparser.**MEDIA_TYPE: Tuple[str]**

```
('application',
 'audio',
 # 'example',  ## preserved for default API
 'font',
 'image',
 'message',
 'model',
 'multipart',
 'text',
 'video')
```

Possible media types.

cfgparser.**API_DICT: Dict[str, *API*]**

Database for API entries.

cfgparser.**API_LOCK: Dict[str, multiprocessing.Lock]**

Database for multiprocessing lock.

cfgparser.**API_INIT: Dict[str, multiprocessing.Value]**

Database for inited flags.

Exceptions

exception cfgparser.**ConfigError**

Bases
Exception

Invalid config.

exception cfgparser.**DefaultNotFoundError**

Bases
ConfigError

The default fallback API for MIME type `example` not found.

exception cfgparser.**ReportNotFoundError**

Bases
ConfigError

The `report` section not found in API.

Common Constants

File location

- Bundled implementation: `source/client/python/const.py`
- Cluster implementation: `cluster/app/source/python/const.py`

const.**ROOT**

Type
`str`

Path to the BroAPT-App framework source codes (absolute path at runtime).

const.**CPU_CNT**

Type
`int`

Environ

- Bundled implementation: *BROAPT_SCAN_CPU*
- Cluster implementation: *BROAPT_APP_CPU*

Number of BroAPT concurrent processes for extracted file analysis. If not provided, then the number of system CPUs will be used.

const.**INTERVAL**

Type
`int`

Environ

- Bundled implementation: *BROAPT_INTERVAL*
- Cluster implementation: *BROAPT_APP_INTERVAL*

Wait interval after processing current pool of extracted files.

const.**MAX_RETRY**

Type
`int`

Retry times for failed commands.

const.**EXIT_SUCCESS = 0**

Type
`int`

Exit code upon success.

const.**EXIT_FAILURE = 1**

Type
int

Exit code upon failure.

const.**LOGS_PATH**

Type
str

Environ
BROAPT_LOGS_PATH

Path to system logs.

const.**DUMP_PATH**

Type
str

Environ
BROAPT_DUMP_PATH

Path to extracted files.

const.**API_ROOT**

Type
str

Environ
BROAPT_API_ROOT

Path to the API root folder.

const.**API_LOGS**

Type
str

Environ
BROAPT_API_LOGS

Path to API detection logs.

const.**API_DICT**

Type
Dict[str, cfgparser.API]

Database for API entries.

See also:

cfgparser.parse

const.**SERVER_NAME_HOST**

Type
str

Environ
BROAPT_NAME_HOST

Hostname of BroAPT-Daemon server.

const.**SERVER_NAME_PORT**

Type
str

Environ
BROAPT_NAME_PORT

Port number of BroAPT-Daemon server.

const.**SERVER_NAME**

Type
str

```
f'http://{SERVER_NAME_HOST}:{SERVER_NAME_PORT}/api/v1.0/scan'
```

URL for BroAPT-Daemon server's scanning API.

const.**DUMP**

Type
str

```
os.path.join(LOGS_PATH, 'dump.log')
```

Path to file system database of processed extracted files.

const.**FAIL**

Type
str

```
os.path.join(LOGS_PATH, 'fail.log')
```

Path to file system database of failed processing extracted files.

const.**FILE_REGEX**

Type
re.Pattern

Availability
bundled implementation

```
re.compile(r'''
    # protocol prefix
    (?P<protocol>DTLS|FTP_DATA|HTTP|IRC_DATA|SMTP|\S+)
    -
    # file UID
    (?P<fuid>F\w+)
    \.
    # PCAP source
    (?P<pcap>.+?)
    \.
    # media-type
    (?P<media_type>
```

(continues on next page)

(continued from previous page)

```
↪application|audio|example|font|image|message|model|multipart|text|video|\S+)
    \.
    # subtype
    (?P<subtype>\S+)
    \.
    # file extension
    (?P<extension>\S+)
''', re.IGNORECASE | re.VERBOSE)
```

Regular expression to match and fetch information from extracted files.

See also:

__main__.FILE_REGEX

const.**MIME_REGEX**

Type
re.Pattern

Availability
bundled implementation

```
re.compile(r'''
    # media-type
    (?P<media_type>
↪application|audio|example|font|image|message|model|multipart|text|video|\S+)
    /
    # subtype
    (?P<subtype>\S+)
''', re.VERBOSE | re.IGNORECASE)
```

Regular expression to match and fetch information from MIME type.

const.**QUEUE_DUMP**

Type
multiprocessing.Queue

Availability
bundled implementation

Teleprocess communication queue for extracted files processing.

Detection Process

File location

- Bundled implementation: source/client/python/scan.py
- Cluster implementation:
  - cluster/app/source/python/scan.py
  - cluster/app/source/python/utils/py

Bundled Implementation

scan.**process**(*entry:* Entry)

Availability
bundled implementation

Process extracted files with detection APIs.

Parameters
entry (Entry) – File to be processed.

scan.**make_env**(*api:* API)

Availability
bundled implementation

Generate a dictionary of environment variables based on API entry.

Parameters
api (API) – API entry from `api.yml`.

Return type
Dict[str, Any]

scan.**make_cwd**(*api:* API, *entry:* Entry | *None* = *None*, *example:* *bool* = *False*)

Availability
bundled implementation

Generate the working directory of API entry.

Parameters

- **api** (API) – API entry from `api.yml`.
- **entry** (Entry) – File to be processed.
- **example** (*bool*) – If using the fallback detection API `example`.

Returns
Path to the working directory.

Return type
str

scan.**init**(*api:* API, *cwd:* *str*, *env:* *Dict[str, Any]*, *mime:* *str*, *uuid:* *str*)

Availability
bundled implementation

Run the initialisation commands of API entry.

Parameters

- **api** (API) – API entry from `api.yml`.
- **cwd** (*str*) – Working directory.
- **env** (*Dict[str, Any]*) – Environment variables.
- **mime** (*str*) – MIME type.
- **uuid** (*str*) – Unique identifier of current scan.

Returns
Exit code (*const.EXIT_SUCCESS* or *const.EXIT_FAILURE*).

Return type
int

scan.**run**(*command: str | List[str]*, *cwd: str = None*, *env: Dict[str, Any] | None = None*, *mime: str = 'example'*, *file: str = 'unknown'*)

Availability
bundled implementation

Run command with provided settings.

Parameters

- **command** (*Union[str, List[str]]*) – Command to execute.
- **cwd** (*str*) – Working dictionary.
- **env** (*Dict[str, Any]*) – Environment variables.
- **mime** (*str*) – MIME type.
- **file** (*str*) – Stem of output log file.

Returns
Exit code (*const.EXIT_SUCCESS* or *const.EXIT_FAILURE*).

Return type
int

scan.**issue**(*mime: str*)

Availability
bundled implementation

Called when the execution of API commands failed.

Parameters
mime (*str*) – MIME type.

Returns
Exit code (*const.EXIT_FAILURE*).

Return type
int

Raises

- ***APIError*** – If `mime` is `example`.
- ***APIWarning*** – If `mime` is **NOT** `example`.

exception scan.**APIWarning**

Bases
Warning

Availability
bundled implementation

Warn if API execution failed.

exception scan.**APIError**

Bases
Exception

Availability
bundled implementation

Error if API execution failed.

Cluster Implementation

process.**process**(*entry:* Entry)

Availability
cluster implementation

See also:

scan.process()

process.**make_env**(*api:* API)

Availability
cluster implementation

See also:

scan.make_env()

process.**make_cwd**(*api:* API, *entry:* Entry | *None* = *None*, *example:* *bool* = *False*)

Availability
cluster implementation

See also:

scan.make_cwd()

process.**init**(*api:* API, *cwd:* *str*, *env:* *Dict[str, Any]*, *mime:* *str*, *uuid:* *str*)

Availability
cluster implementation

See also:

scan.init()

process.**run**(*command:* *str* | *List[str]*, *cwd:* *str* = *None*, *env:* *Dict[str, Any]* | *None* = *None*, *mime:* *str* = *'example'*, *file:* *str* = *'unknown'*)

Availability
cluster implementation

See also:

scan.run()

process.**issue**(*mime:* *str*)

Availability
cluster implementation

See also:

scan.issue()

exception utils.**APIWarning**

Bases
Warning

Availability
cluster implementation

See also:

scan.APIWarning

exception utils.**APIError**

Bases
Exception

Availability
cluster implementation

See also:

scan.APIError

Remote Detection

File location

- Bundled implementation: `source/client/python/scan.py`
- Cluster implementation: `cluster/app/source/python/remote.py`

Bundled Implementation

scan.**remote**(*entry:* Entry, *mime:* *str*, *api:* API)

Availability
bundled implementation

Request the BroAPT-Daemon server to perform *remote* detection.

Parameters

- **entry** (Entry) – Extracted file to be processed.
- **mime** (*str*) – MIME type.
- **api** (API) – API entry from `api.yml`.

Returns
Exit code (*const.EXIT_SUCCESS* or *const.EXIT_FAILURE*).

Return type
int

Cluster Implementation

remote.**remote**(*entry:* Entry, *mime:* *str*, *api:* API)

Availability
cluster implementation

See also:

scan.remote()

Auxiliaries & Utilities

File location

- Bundled implementation: `source/client/python/utils.py`
- Cluster implementation: `cluster/app/source/python/utils.py`

@utils.**suppress**

A decorator that suppresses all exceptions.

utils.**file_lock**(*file:* *str*)

A context lock for file modification with a file system lock.

Parameters
file (*str*) – Filename to be locked in the context.

utils.**temp_env**(*env: Dict[str, Any]*)

A context for temporarily change the current `os.environ`.

Parameters
env (*Dict[str, Any]*) – Environment variables.

utils.**print_file**(*s:* *Any*, *file:* *str*)

Wrapper function to *process*-safely print `s` into `file`.

Parameters

- **s** (*str*) – Content to be printed.
- **file** (*str*) – Filename of output stream.

4.2.2 API Configurations

File location

- Bundled implementation: `source/include/api/`
- Cluster implementation: `cluster/app/include/api/`

As discussed in previous documentation, we provided a YAML configuration file `api.yml` for registering MIME type specific detection methods.

For example, following is the requirements of an API for analysing PDF files (MIME type: `application/pdf`):

- Root: `/api/`
- Target: - MIME type: `application/pdf` - file name: `/dump/application/pdf/test.pdf`

- API: - working directory: `./pdf_analysis` - environment: `ENV_FOO=1`, `ENV_BAR=this is an environment variable`

The configuration section should then be:

```
application:
  ...  # other APIs
  pdf:
    remote: false
    workdir: pdf_analysis
    environ:
      ENV_FOO: 1
      ENV_BAR: this is an environment variable
    install:
      - apt-get update
      - apt-get install -y python python-pip
      - python -m pip install -r requirements.txt
      - rm -rf /var/lib/apt/lists/*
      - apt-get remove -y --auto-remove python-pip
      - apt-get clean
    scripts:
      - ${PYTHON27} detect.py [...]                # refer to /usr/bin/python
      - ...                                        # and some random command
    report: ${PYTHON27} report.py                  # generate final report
```

Important: `report` section is **MANDATORY**.

If `remote` is `true`, then the BroAPT-APP framework will run the corresponding API in the host machine through the BroAPT-Daemon server.

The BroAPT-App framework will work as following:

1. set the following environment variables:
   - per target file
     - `BROAPT_PATH="/dump/application/pdf/test.pdf"`
     - `BROAPT_MIME="application/pdf"`
   - per API configuration
     - `ENV_FOO=1`
     - `ENV_BAR="this is an environment variable"`
2. change the current working directory to `/api/application/pdf/pdf_analysis`
3. if run for the first time, run the following commands:
   - `apt-get update`
   - `apt-get install -y python python-pip`
   - `python -m pip install -r requirements.txt`
   - `rm -rf /var/lib/apt/lists/*`
   - `apt-get remove -y --auto-remove python-pip`
   - `apt-get clean`

4. run the following mid-stage commands:
   - `/usr/bin/python detect.py [...]`
   - ...
5. generate final report: `/usr/bin/python report.py`

Note: The registered MIME types support *shell*-like patterns.

If the API of a specific MIME type is not provided, it will then fallback to the API configuration registered under the special `example` MIME type.

```
## Configuration for API arguments of BroAPT-APP

################################################################################
## Environment (global setup)
##
## Environment variables `${...}` used in API arguments will be translated
## according to the following values.
##
environment:
  # API root path
  API_ROOT: ${BROAPT_API_ROOT}
  # Python 3.6
  PYTHON: /usr/bin/python3.6
  PYTHON36: /usr/bin/python3.6
  PYTHON3: /usr/bin/python3.6
  # Python 2.7
  PYTHON27: /usr/bin/python
  PYTHON2: /usr/bin/python
  # Shell/Bash
  SHELL: /bin/bash

################################################################################
## Example:
##
## - Root: `/api/`
## - Target:
##   - MIME type: `application/pdf`
##   - file name: `/dump/application/pdf/test.pdf`
## - API:
##   - working directory: `./pdf_analysis`
##   - environment: `ENV_FOO=1`, `ENV_BAR=this is an environment variable`
##
## The configuration section should then be:
##
## application:
##   ...  # other APIs
##   pdf:
##     remote: false
##     workdir: pdf_analysis
##     environ:
##       ENV_FOO: 1
```

(continues on next page)

(continued from previous page)

```
##       ENV_BAR: this is an environment variable
##     install:
##       - apt-get update
##       - apt-get install -y python python-pip
##       - python -m pip install -r requirements.txt
##       - rm -rf /var/lib/apt/lists/*
##       - apt-get remove -y --auto-remove python-pip
##       - apt-get clean
##     scripts:
##       - ${PYTHON27} detect.py [...]               # refer to /usr/bin/python
##       - ...                                       # and some random command
##     report: ${PYTHON27} report.py                 # generate final report
##
## BroAPT will work as following:
##
## 1. set the following environment variables
##    # per target file
##    - BROAPT_PATH="/dump/application/pdf/test.pdf"
##    - BROAPT_MIME="application/pdf"
##    # per API configuration
##    - ENV_FOO=1
##    - ENV_BAR="this is an environment variable"
## 2. change the current working directory to
##    `/api/application/pdf/pdf_analysis`
## 3. if run for the first time, run the following commands:
##    - `apt-get update`
##    - `apt-get install -y python python-pip`
##    - `python -m pip install -r requirements.txt`
##    - `rm -rf /var/lib/apt/lists/*`
##    - `apt-get remove -y --auto-remove python-pip`
##    - `apt-get clean`
## 4. run the following mid-stage commands:
##    - `/usr/bin/python detect.py [...]`
##    - `...`
## 5. generate final report:
##    `/usr/bin/python report.py`
##
## NOTE: `report` section is MANDATORY.
##       If `remote` is `true`, then BroAPT will run the
##       corresponding API in the host machine.
##

# APIs for `application` media type
application:
  javascript: &javascript
    ## JaSt
    workdir: ${API_ROOT}/application/javascript/JaSt
    environ:
      JS_LOG: /var/log/bro/tmp/
    install:
      - yum install -y epel-release
      - yum install -y git nodejs
```

(continues on next page)

(continued from previous page)

```
    - test -d ./JaSt/ ||
      git clone https://github.com/Aurore54F/JaSt.git
    - ${PYTHON3} -m pip install
        matplotlib
        plotly
        numpy
        scipy
        scikit-learn
        pandas
    - ${PYTHON3} ./JaSt/clustering/learner.py
        --d ./sample/
        --l ./lables/
        --md ./models/
        --mn broapt-jast
  scripts:
    - ${PYTHON3} ./JaSt/clustering/classifier.py
        --f ${BROAPT_PATH}
        --m ./models/broapt-jast
  report: "false"
octet-stream: &lmd
  ## LMD
  workdir: ${API_ROOT}/application/octet-stream/LMD
  environ:
    LMD_LOG: /var/log/bro/tmp/
  install:
    - yum install -y git which
    - test -d ./linux-malware-detect/ ||
      git clone https://github.com/rfxn/linux-malware-detect.git
    - ${SHELL} install.sh
  report: ${SHELL} detect.sh
vnd.android.package-archive:
  ## AndroPyTool
  remote: true
  workdir: AndroPyTool
  environ:
    # ANDROID_HOME: $HOME/android-sdk-linux
    # PATH: $PATH:$ANDROID_HOME/tools
    # PATH: $PATH:$ANDROID_HOME/platform-tools
    # APK_LOG: /var/log/bro/tmp/
    APK_LOG: /home/traffic/log/bro/tmp/
  install:
    # - ${SHELL} install.sh
    - docker pull alexmyg/andropytool
  # report: ${PYTHON36} detect.py
  report: ${SHELL} detect.sh
vnd.openxmlformats-officedocument: &officedocument
  ## MaliciousMacroBot
  workdir: ${API_ROOT}/application/vnd.openxmlformats-officedocument/
  environ:
    MMB_LOG: /var/log/bro/tmp/
  install:
    - yum install -y git
```

(continues on next page)

(continued from previous page)

```
      - test -d ./MaliciousMacroBot/ ||
        git clone https://github.com/egaus/MaliciousMacroBot.git
      - ${PYTHON36} -m pip install ./MaliciousMacroBot/
      # - rm -rf ./MaliciousMacroBot/
      # - yum erase -y git
      - yum clean -y all
    report: ${PYTHON36} MaliciousMacroBot-detect.py
    shared: officedocument
  msword: *officedocument
  vnd.ms-*: *officedocument
  vnd.openxmlformats-officedocument: *officedocument
  vnd.openxmlformats-officedocument.*: *officedocument
  x-executable:
    ## ELF Parser
    remote: true
    environ:
      # ELF_LOG: /var/log/bro/tmp/
      ELF_LOG: /home/traffic/log/bro/tmp/
      ELF_SCORE: 100
    workdir: ELF-Parser
    install:
      - docker build --tag elfparser:1.4.0 --rm .
      # - yum install -y git cmake make boost-devel gcc gcc-g++
      # - test -d ./elfparser/ ||
      #   git clone https://github.com/jacob-baines/elfparser.git
      # - ${SHELL} build.sh
      # - rm -rf ./elfparser/
      # # - yum erase -y git cmake make
      # - yum clean -y all
    report: ${SHELL} detect.sh

# APIs for `audio` media type
audio:

# Default API for missing MIME types
example:
  environ:
    ## sleep interval
    VT_INTERVAL: 30
    ## max retry for report
    VT_RETRY: 10
    ## percentage of positive threshold
    VT_PERCENT: 50
    ## VT API key
    #VT_API: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    ## path to VT file scan reports
    VT_LOG: /var/log/bro/tmp/
  report: ${PYTHON36} virustotal.py || exit 0 # always EXIT_SUCCESS

# APIs for `font` media type
font:
```

(continues on next page)

(continued from previous page)

```
# APIs for `image` media type
image:

# APIs for `message` media type
message:

# APIs for `model` media type
model:

# APIs for `multipart` media type
multipart:

# APIs for `text` media type
text:
  html: *lmd
  javascript: *javascript
  x-c: *lmd
  x-perl: *lmd
  x-php: *lmd

# APIs for `video` media type
video:
```

```
## Configuration for API arguments of BroAPT-APP

################################################################################
## Environment (global setup)
##
## Environment variables `${...}` used in API arguments will be translated
## according to the following values.
##
environment:
  # API root path
  API_ROOT: ${BROAPT_API_ROOT}
  # Python 3.6
  PYTHON: /usr/bin/python3.6
  PYTHON36: /usr/bin/python3.6
  PYTHON3: /usr/bin/python3.6
  # Python 2.7
  PYTHON27: /usr/bin/python
  PYTHON2: /usr/bin/python
  # Shell/Bash
  SHELL: /bin/bash

################################################################################
## Example:
##
## - Root: `/api/`
## - Target:
##   - MIME type: `application/pdf`
##   - file name: `/dump/application/pdf/test.pdf`
## - API:
```

(continues on next page)

(continued from previous page)

```
##   - working directory: `./pdf_analysis`
##   - environment: `ENV_FOO=1`, `ENV_BAR=this is an environment variable`
##
## The configuration section should then be:
##
## application:
##   ...  # other APIs
##   pdf:
##     remote: false
##     workdir: pdf_analysis
##     environ:
##       ENV_FOO: 1
##       ENV_BAR: this is an environment variable
##     install:
##       - apt-get update
##       - apt-get install -y python python-pip
##       - python -m pip install -r requirements.txt
##       - rm -rf /var/lib/apt/lists/*
##       - apt-get remove -y --auto-remove python-pip
##       - apt-get clean
##     scripts:
##       - ${PYTHON27} detect.py [...]               # refer to /usr/bin/python
##       - ...                                       # and some random command
##     report: ${PYTHON27} report.py                 # generate final report
##
## BroAPT will work as following:
##
## 1. set the following environment variables
##    # per target file
##    - BROAPT_PATH="/dump/application/pdf/test.pdf"
##    - BROAPT_MIME="application/pdf"
##    # per API configuration
##    - ENV_FOO=1
##    - ENV_BAR="this is an environment variable"
## 2. change the current working directory to
##    `/api/application/pdf/pdf_analysis`
## 3. if run for the first time, run the following commands:
##    - `apt-get update`
##    - `apt-get install -y python python-pip`
##    - `python -m pip install -r requirements.txt`
##    - `rm -rf /var/lib/apt/lists/*`
##    - `apt-get remove -y --auto-remove python-pip`
##    - `apt-get clean`
## 4. run the following mid-stage commands:
##    - `/usr/bin/python detect.py [...]`
##    - `...`
## 5. generate final report:
##    `/usr/bin/python report.py`
##
## NOTE: `report` section is MANDATORY.
##       If `remote` is `true`, then BroAPT will run the
##       corresponding API in the host machine.
```

(continues on next page)

(continued from previous page)

```
##

# APIs for `application` media type
application:
  javascript: &javascript
    ## JaSt
    workdir: ${API_ROOT}/application/javascript/JaSt
    environ:
      JS_LOG: /var/log/bro/tmp/
    install:
      - apt-get update
      - apt-get install -y --no-install-recommends git nodejs
      - test -d ./JaSt/ ||
        git clone https://github.com/Aurore54F/JaSt.git
      - ${PYTHON3} -m pip install
          matplotlib
          plotly
          numpy
          scipy
          scikit-learn
          pandas
      - ${PYTHON3} ./JaSt/clustering/learner.py
          --d ./sample/
          --l ./lables/
          --md ./models/
          --mn broapt-jast
    scripts:
      - ${PYTHON3} ./JaSt/clustering/classifier.py
          --f ${BROAPT_PATH}
          --m ./models/broapt-jast
    report: "false"
  octet-stream: &lmd
    ## LMD
    workdir: ${API_ROOT}/application/octet-stream/LMD
    environ:
      LMD_LOG: /var/log/bro/tmp/
    install:
      - apt-get install -y --no-install-recommends git
      - test -d ./linux-malware-detect/ ||
        git clone https://github.com/rfxn/linux-malware-detect.git
      - ${SHELL} install.sh
    report: ${SHELL} detect.sh
  vnd.android.package-archive:
    ## AndroPyTool
    remote: true
    workdir: AndroPyTool
    environ:
      # ANDROID_HOME: $HOME/android-sdk-linux
      # PATH: $PATH:$ANDROID_HOME/tools
      # PATH: $PATH:$ANDROID_HOME/platform-tools
      # APK_LOG: /var/log/bro/tmp/
      APK_LOG: /home/traffic/log/bro/tmp/
```

(continues on next page)

(continued from previous page)

```
    install:
      # - ${SHELL} install.sh
      - docker pull alexmyg/andropytool
    # report: ${PYTHON36} detect.py
    report: ${SHELL} detect.sh
  vnd.openxmlformats-officedocument: &officedocument
    ## MaliciousMacroBot
    workdir: ${API_ROOT}/application/vnd.openxmlformats-officedocument/
    environ:
      MMB_LOG: /var/log/bro/tmp/
    install:
      - apt-get install -y --no-install-recommends git
      - test -d ./MaliciousMacroBot/ ||
        git clone https://github.com/egaus/MaliciousMacroBot.git
      - ${PYTHON36} -m pip install ./MaliciousMacroBot/
    report: ${PYTHON36} MaliciousMacroBot-detect.py
    shared: officedocument
  msword: *officedocument
  vnd.ms-*: *officedocument
  vnd.openxmlformats-officedocument: *officedocument
  vnd.openxmlformats-officedocument.*: *officedocument
  x-executable:
    ## ELF Parser
    remote: false
    environ:
      # ELF_LOG: /var/log/bro/tmp/
      ELF_LOG: /home/traffic/log/bro/tmp/
      ELF_SCORE: 100
    workdir: ELF-Parser
    install:
      - apt-get update
      - apt-get install -y --no-install-recommends \
          cmake \
          g++ \
          gcc \
          git \
          libboost-all-dev \
          make
      - test -d ./elfparser/ ||
        git clone https://github.com/jacob-baines/elfparser.git
      - ${SHELL} build.sh
    report: ${SHELL} detect.sh

# APIs for `audio` media type
audio:

# Default API for missing MIME types
example:
  environ:
    ## sleep interval
    VT_INTERVAL: 30
    ## max retry for report
```

(continues on next page)

(continued from previous page)

```
    VT_RETRY: 10
    ## percentage of positive threshold
    VT_PERCENT: 50
    ## VT API key
    #VT_API: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    ## path to VT file scan reports
    VT_LOG: /var/log/bro/tmp/
  report: ${PYTHON36} virustotal.py || exit 0 # always EXIT_SUCCESS

# APIs for `font` media type
font:

# APIs for `image` media type
image:

# APIs for `message` media type
message:

# APIs for `model` media type
model:

# APIs for `multipart` media type
multipart:

# APIs for `text` media type
text:
  html: *lmd
  javascript: *javascript
  x-c: *lmd
  x-perl: *lmd
  x-php: *lmd

# APIs for `video` media type
video:
```

Caution: For bundled implementation, the runtime of *local* APIs are in the CentOS 7 Docker container.

For cluster implementation, the runtime of *local* APIs are in the Ubuntu 16.04 Docker container.

4.2.3 Wrapper Scripts

For the Docker container, we have created some Shell/Bash wrapper scripts to make the life a little bit better.

Bundled Implementation

File location
source/client/init.sh

As the BroAPT-App framework is already integrated into the source codes, there's no need to another wrapper script to start the BroAPT-App framework. It shall be run directly after the BroAPT-Core framework.

```
#!/usr/bin/env bash

set -aex

# change curdir
cd /broapt

# load environs
if [ -f .env ] ; then
    source .env
fi

# compose Bro scripts
/usr/bin/python3.6 python/compose.py

# run scripts
/usr/bin/python3.6 python $@

# sleep
sleep infinity
```

Cluster Implementation

File location
cluster/app/source/init.sh

```
#!/usr/bin/env bash

set -aex

# change cwd
cd /source

# load environs
if [ -f .env ] ; then
    source .env
fi

# run scripts
/usr/bin/python3.6 python
```

(continues on next page)

(continued from previous page)

```
# sleep
sleep infinity
```

4.3 BroAPT-Daemon Server

The BroAPT-Daemon server is the main entry and watchdog for the BroAPT system. For more information about the server, please refer to previous documentation at *BroAPT-App Detection Framework*.

4.3.1 Module Entry

File location

- Bundled implementation: `source/server/python/__init__.py`
- Cluster implementation: `cluster/daemon/python/__init__.py`

This file merely modifies the `sys.path` so that we can import the Python modules as if from the top level.

4.3.2 System Entrypoint

File location

- Bundled implementation: `source/server/python/__main__.py`
- Cluster implementation: `cluster/daemon/python/__main__.py`

This file wraps the whole system and make the `python` folder callable as a module where the `__main__.py` will be considered as the entrypoint.

run()

Start the Flask application and Docker watchdog.

4.3.3 Command Line Interface

File location

- Bundled implementation: `source/server/python/cli.py`
- Cluster implementation: `cluster/daemon/python/cli.py`

For options and configuration details, please refer to *configuration* documentations.

parse_args()

Parse command line arguments.

Returns
Parsed command line arguments.

Return type
argparse.Namespace

parse_env()

Parse provided *dotenv* files for the command line argument parser as default values.

Returns
Parsed *dotenv* values.

Return type
Dict[str, Any]

4.3.4 Docker Watchdog

File location

- Bundled implementation: `source/server/python/compose.py`
- Cluster implementation: `cluster/daemon/python/compose.py`

This module provides a handy way to always keep the underlying BroAPT system in Docker containers running.

compose.**docker_compose**()

A context to manager Docker containers. This function will start *watch_container()* as a background process.

Note: When start, the function will start the Docker containers through *start_container()*.

Before exit, the function will toggle the value of *UP_FLAG* to `False` and wait for the process to exit. And gracefully stop the Docker containers through *stop_container()*.

---

compose.**watch_container**()

Supervise the status of Docker containers while the system is running, i.e. *UP_FLAG* is `True`.

Raises
ComposeWarning – If fail to poll status of Docker containers.

compose.**start_container**()

Start Docker container using Docker Compose in *detached* mode.

compose.**stop_container**()

Stop Docker container gracefully using Docker Compose, and clean up Docker caches.

compose.**flask_exit**(*signum: signal.Signals | None = None*, *frame: types.FrameType | None = None*)

Flask exit signal handler. This function is registered as handler for *const.KILL_SIGNAL* through *register()*.

compose.**register**()

Register *flask_exit()* as signal handler of *const.KILL_SIGNAL*.

compose.**UP_FLAG = multiprocessing.Value('B', True)**

If the BroAPT system is actively running.

exception compose.**ComposeWarning**

Bases
Warning

Warn if fail to poll status of Docker containers.

4.3.5 Common Constants

File location

- Bundled implementation: `source/server/python/const.py`
- Cluster implementation: `cluster/daemon/python/const.py`

`const.`**KILL_SIGNAL**

Type
`int`

Environ
BROAPT_KILL_SIGNAL

Daemon kill signal.

`const.`**SERVER_NAME_HOST**

Type
`str`

Environ
BROAPT_SERVER_HOSTs

The hostname to listen on.

`const.`**SERVER_NAME_PORT**

Type
`int`

Environ
BROAPT_SERVER_PORT

`const.`**DOCKER_COMPOSE**

Type
`str`

Environ
BROAPT_DOCKER_COMPOSE

Path to BroAPT's compose file.

`const.`**DUMP_PATH**

Type
`str`

Environ
BROAPT_DUMP_PATH

Path to extracted files.

`const.`**LOGS_PATH**

Type
`str`

Environ
BROAPT_LOGS_PATH

Path to log files.

const.**API_LOGS**

Type
str

Environ
BROAPT_API_LOGS

Path to API runtime logs.

const.**API_ROOT**

Type
str

Environ
BROAPT_API_ROOT

Path to detection APIs.

const.**INTERVAL**

Type
float

Environ
BROAPT_INTERVAL

Sleep interval.

const.**MAX_RETRY**

Type
str

Environ
BROAPT_MAX_RETRY

Command retry.

const.**EXIT_SUCCESS = 0**

Type
int

Exit code upon success.

const.**EXIT_FAILURE = 1**

Type
int

Exit code upon failure.

const.**FILE**

Type
str

```
os.path.join(LOGS_PATH, 'dump.log')
```

Path to file system database of processed extracted files.

const.**FAIL**

Type
str

```
os.path.join(LOGS_PATH, 'fail.log')
```

Path to file system database of failed processing extracted files.

4.3.6 Flask Application

File location

- Bundled implementation: `source/server/python/daemon.py`
- Cluster implementation: `cluster/daemon/python/daemon.py`

URL Routing

daemon.**root**()

Route
/api

Methods
GET

Display help message *__help__*.

daemon.**help_**()

Route
/api/v1.0

Methods
GET

Display help message *HELP_v1_0*.

daemon.**list_**()

Route
/api/v1.0/list

Methods
GET

List of detection process information.

0. Information of running processes from *RUNNING*:

```
{
  "id": "...",
  "initied": null,
  "scanned": true,
  "reported: null,
  "deleted": false
}
```

1. Information of finished processes from *SCANNED*:

   - If the process exited on success:

   ```
   {
     "id": "...",
     "initied": null,
     "scanned": true,
     "reported: true,
     "deleted": false
   }
   ```

   - If the process exited on failure:

   ```
   {
     "id": "...",
     "initied": null,
     "scanned": true,
     "reported: false,
     "deleted": false
   }
   ```

`get_none()`

Route
`/api/v1.0/report`

Methods
`GET`

Display help message:

```
ID Required: /api/v1.0/report/<id>
```

`get(id_: str)`

Route
`/api/v1.0/report/<id>`

Methods
`GET`

Fetch detection status of `id_`.

0. If `id_` in *RUNNING*:

   ```
   {
     "id": "...",
     "initied": null,
     "scanned": false,
     "reported: null,
     "deleted": false
   }
   ```

1. If `id_` in *SCANNED*:

   - If the process exited on success:

```
{
  "id": "...",
  "initied": null,
  "scanned": true,
  "reported: true,
  "deleted": false
}
```

- If the process exited on failure:

```
{
  "id": "...",
  "initied": null,
  "scanned": true,
  "reported: false,
  "deleted": false
}
```

2. If `id_` not found, raises `404 Not Found` with *id_not_found()*.

daemon.**scan**()

Route
`/api/v1.0/scan`

Methods
`POST`

Perform *remote* detection on target file.

The `POST` data should be a JSON object with following fields:

Parameters

- **name** (*string*) – path to the extracted file
- **mime** (*string*) – MIME type
- **uuid** (*string*) – unique identifier
- **report** (*string | string[]*) – report generation commands
- **shared** (*string*) – shared detection API identifier
- **inited** (*boolean*) – API initialised
- **workdir** (*string*) – working directory
- **environ** (*object*) – environment variables
- **install** (*string | string[]*) – initialisation commands
- **scripts** (*string | string[]*) – detection commands

If **NO** JSON data provided, raises `400 Bad Request` with *invalid_info()*.

After performing detection *process.process()* on the target file, returns a JSON object containing detection report:

0. If detection exits on success:

```
{
  "id": "...",
  "initied": true,
  "scanned": true,
  "reported: true,
  "deleted": false
}
```

1. If detection exists on failure:

   - If detection fails when initialising:

   ```
   {
     "id": "...",
     "initied": false,
     "scanned": true,
     "reported: false,
     "deleted": false
   }
   ```

   - If detection fails when processing:

   ```
   {
     "id": "...",
     "initied": true,
     "scanned": true,
     "reported: false,
     "deleted": false
   }
   ```

delete_none()

Route
/api/v1.0/delete

Methods
DELETE

Display help message:

```
ID Required: /api/v1.0/delete/<id>
```

delete(*id_: str*)

Route
/api/v1.0/delete/<id>

Methods
DELETE

Delete detection status of id_.

0. If id_ in *RUNNING*:

```
{
  "id": "...",
  "initied": null,
```

(continues on next page)

(continued from previous page)

```
  "scanned": false,
  "reported: null,
  "deleted": true
}
```

1. If `id_` in *SCANNED*:

   - If the process exited on success:

   ```
   {
     "id": "...",
     "initied": null,
     "scanned": true,
     "reported: true,
     "deleted": true
   }
   ```

   - If the process exited on failure:

   ```
   {
     "id": "...",
     "initied": null,
     "scanned": true,
     "reported: false,
     "deleted": true
   }
   ```

2. If `id_` not found:

```
{
  "id": "...",
  "initied": null,
  "scanned": null,
  "reported: null,
  "deleted": true
}
```

Error Handlers

daemon.**invalid_id**(*error: Exception*)

Handler of `ValueError`.

```
{
  "status": 400,
  "error": "...",
  "message": "invalid ID format"
}
```

daemon.**invalid_info**(*error: Exception*)

Handler of `400 Bad Request` and `KeyError`.

```
{
  "status": 400,
  "error": "...",
  "message": "invalid info format"
}
```

daemon.**id_not_found**(*error: Exception*)

Handler of `404 Not Found`.

```
{
  "status": 404,
  "error": "...",
  "message": "ID not found"
}
```

Dataclasses

class daemon.**INFO**

A dataclass for requested detection API information.

name: str

Path to the extracted file.

uuid: str

Unique identifier of current process.

mime: str

MIME type.

report: str

Report generation command.

inited: manager.Value

Initied flag.

locked: multiprocessing.Lock

Multiprocessing runtime lock.

workdir: str

API working directory.

environ: Dict[str, Any]

API runtime environment variables.

install: List[str | List[str]]

List of installation commands.

scripts: List[str | List[str]]

List of detection commands.

Constants

daemon.**app = flask.Flask(__name__)**

Flask application.

daemon.**HELP_v1_0**: str

```
BroAPT Daemon APIv1.0 Usage:

- GET    /api/v1.0/list
- GET    /api/v1.0/report/<id>
- POST   /api/v1.0/scan data={"key": "value"}
- DELETE /api/v1.0/delete/<id>
```

daemon.**__help__**: str

```
BroAPT Daemon API Usage:

# v1.0

- GET    /api/v1.0/list
- GET    /api/v1.0/report/<id>
- POST   /api/v1.0/scan data={"key": "value"}
- DELETE /api/v1.0/delete/<id>
```

daemon.**manager = multiprocessing.Manager()**

Multiprocessing manager instanace.

daemon.**RUNNING = manager.list()**

Type
List[uuid.UUID]

List of running detection processes.

daemon.**SCANNED = manager.dict()**

Type
Dict[uuid.UUID, bool]

Record of finished detection processes and exit on success.

daemon.**APILOCK = manager.dict()**

Type
Dict[str, multiprocessing.Lock]

Record of API multiprocessing locks.

daemon.**APIINIT = manager.dict()**

Type
Dict[str, multiprocessing.Value]

Record of API initialised flags.

4.3.7 Detection Process

File location

- Bundled implementation: `source/server/python/process.py`
- Cluster implementation: `cluster/daemon/python/process.py`

process.**process**(*info:* INFO)

Process extracted files with detection information.

Parameters
info (INFO) – File to be processed.

Returns
If detection process exit on success.

Return type
bool

process.**make_env**(*info:* INFO)

Generate a dictionary of environment variables based on request information.

Parameters
info (INFO) – Detection request information.

Return type
Dict[str, Any]

process.**make_cwd**(*info:* INFO)

Generate the working directory of detection information.

Parameters
info (INFO) – Detection request information.

Returns
Path to the working directory.

Return type
str

process.**init**(*info:* INFO)

Run the initialisation commands of detection information.

Parameters
info (INFO) – Detection request information.

Returns
Exit code (*const.EXIT_SUCCESS* or *const.EXIT_FAILURE*).

Return type
int

process.**run**(*command: str | List[str]*, *info:* INFO, *file: str = 'unknown'*)

Run command with provided settings.

Parameters

- **command** (*Union[str, List[str]]*) – Command to execute.
- **info** (INFO) – Detection request information.
- **file** (*str*) – Stem of output log file.

Returns
Exit code (*const.EXIT_SUCCESS* or *const.EXIT_FAILURE*).

Return type
int

4.3.8 Auxiliaries & Utilities

File location

- Bundled implementation: `source/server/python/util.py`
- Cluster implementation: `cluster/daemon/python/util.py`

`@utils.`**`suppress`**
A decorator that suppresses all exceptions.

`utils.`**`file_lock`**(*file: str*)
A context lock for file modification with a file system lock.

Parameters
file (*str*) – Filename to be locked in the context.

`utils.`**`print_file`**(*s: Any*, *file: str*)
Wrapper function to *process*-safely print `s` into `file`.

Parameters

- **s** (*str*) – Content to be printed.
- **file** (*str*) – Filename of output stream.

`utils.`**`temp_env`**(*env: Dict[str, Any]*)
A context for temporarily change the current `os.environ`.

Parameters
env (*Dict[str, Any]*) – Environment variables.

For deployment issues, please refer to *quickstart*.

4.4 Miscellaneous & Auxiliary

4.4.1 MIME-Extension Mappings

Generate Mappings

File location

- Bundled implementation: `source/utils/mime2ext.py`
- Cluster implementation: `cluster/utils/mime2ext.py`

Note: This script support all version since Python 2.7.

BROAPT_FORCE_UPDATE

Type
`bool`

Default
`False`

Set the environment variable to `True` if you wish to update existing mappings; otherwise, it will only add mappings of new MIME types.

The script fetch the MIME types from IANA registries and try to automatically match them with the file extensions through `mimetypes` database. It will then dump the mappings to corresponding `file-extensions.bro` as discussed in the *documentation*.

Should there be an unknown MIME type, it will prompt for user to type in the corresponding file extensions.

Fix Missing Mappings

File location

- Bundled implementation: `source/utils/fix-missing.py`
- Cluster implementation: `cluster/utils/fix-missing.py`

Note: This script support all version since Python 2.7.

BROAPT_LOGS_PATH

Type
`str` (path)

Default
`/var/log/bro/`

Path to system logs.

In the BroAPT system, when encountering a MIME type not present in the `file-extensions.bro` database, it will record such MIME type into a log file under the log path *`const.LOGS_PATH`*, named `processed_mime.log`.

The script will read the log file and try to update the `file-extensions.bro` database with these found-missing MIME types.

4.4.2 Bro Script Composers

HTTP Method Registry

File location
`source/utils/http-methods.py`

Note: This script support all version since Python 2.7.

As discussed in *BroAPT-Core Extration Framework*, we have introduced full HTTP methods registry to the BroAPT system in Bro script `sites/const/http-methods.bro`.

The script will read the IANA registries and update the builtin `HTTP::http_methods` with the fetched data.

HTTP Message Headers

File location
`source/utils/http-header-names.py`

Note: This script support all version since Python 2.7.

As discussed in *BroAPT-Core Extration Framework*, we have introduced full HTTP message header registry to the BroAPT system in Bro script `sites/const/http-header-names.bro`.

The script will read the IANA registries and update the builtin `HTTP::header_names` with the fetched data.

FTP Commands & Extensions

File location
`source/utils/ftp-commands.py`

Note: This script support all version since Python 2.7.

As discussed in *BroAPT-Core Extration Framework*, we have introduced full FTP commands ands extensions registry to the BroAPT system in Bro script `sites/const/ftp-commands.bro`.

The script will read the IANA registries and update the builtin `FTP::logged_commands` with the fetched data.

4.5 System Runtime

The whole BroAPT folder in the Docker container (**of bundled implementation**) at runtime would be like:

```
# project root
/broapt/
├── # entrypoint wrapper script
├── init.sh
├── # Python source codes
├── python
│   ├── # setup PYTHONPATH
│   ├── __init__.py
│   ├── # entry point
│   ├── __main__.py
│   ├── # config parser
│   ├── cfgparser.py
│   ├── # Bro script composer
│   ├── compose.py
│   ├── # global constants
│   ├── const.py
│   ├── # Bro log parser
│   ├── logparser.py
│   ├── # BroAPT-Core logic
│   ├── process.py
│   ├── # multiprocessing support
│   ├── remote.py
```

(continues on next page)

(continued from previous page)

```
│   │   # BroAPT-App logic
│   ├── scan.py
│   │   # Python hooks
│   ├── sites
│   │   │   # register hooks
│   │   ├── __init__.py
│   │   └── ...
│   │   # utility functions
│   └── utils.py
│   # Bro source scripts
└── scripts
    │   # load FileExtraction module
    ├── __load__.bro
    │   # configurations
    ├── config.bro
    │   # MIME-extension mappings
    ├── file-extensions.bro
    │   # protocol hooks
    ├── hooks/
    │   │   # extract DTLS
    │   ├── extract-dtls.bro
    │   │   # extract FTP_DATA
    │   ├── extract-ftp.bro
    │   │   # extract HTTP
    │   ├── extract-http.bro
    │   │   # extract IRC_DATA
    │   ├── extract-irc.bro
    │   │   # extract SMTP
    │   └── extract-smtp.bro
    │   # core logic
    ├── main.bro
    │   # MIME hooks
    ├── plugins/
    │   │   # extract all files
    │   ├── extract-all-files.bro
    │   │   # extract by BRO_MIME
    │   ├── extract-white-list.bro
    │   │   # generated scripts by BRO_MIME
    │   └── ...
    │   # site functions by user
    └── sites/
        │   # load site functions
        ├── __load__.bro
        └── ...
```

where `/broapt/python/sites` is the path for custom Python hooks and `/broapt/scripts/sites/` is the path for custom Bro scripts.

And most importantly, the very entrypoint for the whole BroAPT system is as following:

```
#!/usr/bin/env bash

set -aex
```

(continues on next page)

(continued from previous page)

```
# change curdir
cd /broapt

# load environs
if [ -f .env ] ; then
    source .env
fi

# compose Bro scripts
/usr/bin/python3.6 python/compose.py

# run scripts
/usr/bin/python3.6 python $@

# sleep
sleep infinity
```

0. The script will first change the current working directory to the root path `/broapt/`.
1. If there is a `.env` *dotenv* file for environment variables configuration, it will be loaded and saved into current runtime scope (`set -a`).
2. Generate Bro scripts based on environment variables.
3. Start the main application, i.e. BroAPT-Core and BroAPT-App frameworks.

4.6 Developer Notes

Since the BroAPT system was not intended for packaging and distribution, we didn't provide a `setup.py` to wrap everything as a `broapt` module. However, in a quite *hacky* way, we injected the `sys.path` import path, so that we can directly import the files as if they're at top levels.

As you can see in the `/broapt/python/sites/__init__.py`, i.e. the *module* entry of Python hooks is as following:

```
# -*- coding: utf-8 -*-
# pylint: disable=all

###############################################################################
# site customisation
import os
import sys

sys.path.insert(0, os.path.dirname(os.path.realpath(__file__)))
sys.path.insert(0, os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
###############################################################################

from extracted_files import generate_log as info_log
from http_parser import generate as http_log, close as http_log_exit

# log analysis hook list
HOOK = [
```

(continues on next page)

(continued from previous page)

```
   http_log,
   info_log,
]

# exit hooks
EXIT = [
   http_log_exit,
]
```

where `extracted_files` refers to `/broapt/python/sites/extracted_files.py` and `http_parser` refers to `/broapt/python/sites/http_parser.py`.

You may have noticed the lines in *site customisation* modified the `sys.path` import path so that we don't need to worry about importing stuff from the BroAPT Python source codes.

If you wish to use auxiliary functions and module constants from the main application, then you can still import them as if from the top level:

```
# path to logs from module constants
from const import LOGS_PATH
# Bro log parsing utilities
from logparser import parse
# auxiliary functions for BroAPT
from utils import is_nan, print_file
```

Cybersecurity has long been a significant subject under discussion. With rapid evolution of new cyber attack methods, the threat of Internet is becoming more and more intense. Advanced persistent threat (APT) has become a main source of cybersecurity events. It is now even more important to identify and classify network traffic by direct analysis on the traffic itself in an accurate and timely manner.

We hereby describe BroAPT system, an APT detection system based on Bro IDS (old name at time of implementation, now known as Zeek IDS). The system monitors APT based on comprehensive analysis of the network traffic. It is granted with high performance and extensibility. It can reassemble then extract files transmitted in the traffic, analyse and generate log files in real-time; it can also classify extracted files through targeted malicious file detection configuration; and it detects APT attacks based on analysis of the log files generated by the system itself.

The BroAPT system consists of two major parts. One is the core functions. This part runs in a Docker container, which currently is based on CentOS 7 image. The core functions can be described by two different components: an extraction framework BroAPT-Core and a detection framework BroAPT-App. The other is the command line interface (CLI) and a daemon server BroAPT-Daemon, which is a RESTful API server based on Flask framework. This part runs on the host machine of the Docker container.

CLI is the entrypoint for the whole BroAPT system. When running, the CLI configures the daemon server and bring it up, then start the Docker container with core functions. Within BroAPT-Core extraction framework, it will read in a PCAP file and process it with Bro IDS, which will reassemble then extract files transmitted by the traffic and generate log files from its logging system. Afterwards, BroAPT-App detection framework will take the extracted file, parse it's file name to extract MIME type information of this file. Then the framework will fetch specific detection API of such MIME type and process it to detect if the file is malicious. If needed, the framework will generate a request to BroAPT-Daemon server to process a remote (privileged) detection on such file.

Of BroAPT-Core extraction framework, it mainly has three steps. First, file check. The system will scan for new PCAP files and send them to the BroAPT-Core extraction framework. Second, Bro analysis. The system will process the PCAP with file extraction scripts, reassemble then extract files transmitted through the traffic. The extraction can be grouped by MIME type of files or application layer protocol which transmitted the file. Also, the user may load external Bro scripts as site functions to process along with the main extraction scripts. Third, post-processing and cross-analysis. After processing the PCAP file with Bro IDS, the system will have several extracted files and a bunch of

log files. Besides those standard Bro logs, there will be logs defined by the site functions and generated by the logging system of Bro IDS. Then the system, by default, will generate connection information of the extracted files through Bro logs, which includes timestamp, source and destination, MIME type, as well as hash values. Plus, the user may also register Python hooks to the system, as they will be called every time a PCAP is processed. These hooks can to used to provide further investigation upon the logs generated by Bro IDS.

To work along with Bro intrusion detection system (IDS), the system is implemented in a multi-processing manner. Since CPython's multi-threading is not working as expected – cannot perform parallel processing – we implemented BroAPT system with full support of multi-processing to accelerate the main processing logic. Synchronised queues are used to communicate and coordinate processes within the system: in BroAPT-Core extraction framework, we used a queue to send basic information about the extracted files to BroAPT-App detection framework, and another queue to procede Python hooks with the generated log files.

Currently, we have introduced several site functions and Python hooks to BroAPT system. There are six bunches of Bro scripts. Constant definitions for common application layer protocols, such as HTTP and FTP, these constants are fetched from IANA registry. Extend standard Bro log http.log with new entry of COOKIE information and data in POST request. Calculate hash values of all files transmitted through network traffic. And two Bro modules to perform phishing emails based on cross-analysis of SMTP and HTTP traffic. The Python hook function currently included is to parse http.log then extract information of HTTP connections and generate a new log file.

As for BroAPT-App detection framework, we genetically designed the client-server remote detection framework based on the support of BroAPT-Daemon server. Briefly, the BroAPT-App detection framework will take the extracted files as input source. The system will perform file check to extract information from it. These information includes path to the file, MIME type and unique identifier (UID) of such file, etc. Then the system will parse an API configuration file to obtain a mapping of MIME type specific malicious file detection APIs. Based on the MIME type we had from the file, the system will perform APT detection with the selected API. When detection, the system will firstly prepare the working environment according to the API configuration: it will assign environment variables, change working directory accordingly, expand variables defined in scripts then execute installations scripts. Afterwards, the system will execute detection scripts, then report generation script to generate detection results for the target file. If a remote detection is required, the system will prepare the request data, then post it to the BroAPT-Daemon server running on the host machine. The BroAPT-Daemon server will process the detection ibid.

Speaking of installation, we introduced several attributes to manage and avoid resource competition. We used a shared memory space to indicate whether such API has been proceded with installation. This indicator will avoid reinstallation of APIs. It is shared with all MIME type specific APIs that sharing the same detection process, not just processes using the same API. Additionally, we have a synchronised process lock to prevent parallel installation for the same APIs. However, considering the APIs might fail due to network connection issue, we will try to rerun the script if it fails.

We have by far introduced, six different APIs targeted for dozens of MIME types. We used VirusTotal as the basic general detection method for BroAPT, which will detect any MIME types that have no registered API; VirusTotal aggregates many antivirus products and online scan engines to check for ciruses that the user's own antivirus may have missed, or to verify against any false positives. We used AndroPyTool to detect APK files (MIME type: `application/vnd.android.package-archive`); AndroPyTool is a tool for extracting statis and dynamic features from Android APKs, which combines different well-known Android application analysis tools such as DroidBox, FlowDroid, Strace, AndroGuard or VirusTotal analysis. We used MaliciousMacroBot to detect Office documents (MIME type: `application/vnd.openxmlformats-officedocument`, or `application/msword`, `application/vnd.ms-excel`, `application/vnd.ms-powerpoint`, etc.); MaliciousMacroBot provides a powerful malicious file triage tool through clever feature engineering and applied machine learning techniques like Random Forest and TF-IDF. We used ELF Parser to detect Linux ELF binaries (MIME type: `application/x-executable`); ELF Parser is a static ELF analysis tool to quickly determine the capabilities of an ELF binary through statis analysis. We used LMD to detect other common Linux exploitable files (MIME type: `application/octet-stream`, `text/html`, `text/x-c`, `text/x-perl`, `text/x-php`, etc.); LMD is a malware scanner for Linux systems based on threat data from network edge intrusion detection systems to extract malware that is actively being used in attacks and generates signatures for detection. And we used JaSt to detect JavaScript files (MIME type: `application/javascript` or `text/javascript`); JaSt is a tool to syntactically detect malicious (obfuscated) JavaScript files based on machine learning and clustering algorithms.

As described above, BroAPT is an APT detection system based on Bro IDS with high extensibility and compatibility with high-speed traffic. We tested BroAPT system with real-time traffic collected from the network edge of a college. The system will extract all targeted files from an approximately 35G PCAP file within one minute. And the Bro site functions introduced within BroAPT-Core extraction framework has no significant impact on performance of the system, whilst the Python hook functions will smoothly work along and generate new log files as it intended to. Also, the detection APIs we used in BroAPT-App detection system has proved that they are working perfectly with reasonable false-positive rates. In a word, the BroAPT system is working as expected in the real network environment.

However, besides the implementation above, we have tried several other implementations during the project. We used pure Python scripts based on PyPCAPKit (a multi-engine PCAP file analysis tool) with supoort of DPKT to reassemble and extract files transmitted through the traffic, but the process efficiency was not quite good. Not to mension hybrid implementation with Bro scripts logging TCP traffic data and Python or C/C++ programs to reassembly then extract the traffic, and the miserable pure Bro implementation of TCP reassembly. At last, File Analysis framework of Bro IDS proved its worthiness to the BroAPT system. And thus we adopted the current implementation.

Although our research on APT detection is quite preceding, the BroAPT system utilised Bro IDS and works as an APT detection system which is compatible with high-speed network traffic. The system has been proved in practical scenarios, and is the basis of follow-up researches on APT detection.

For more information, please refer to the `Graduation Thesis` of BroAPT (in Chinese).

CHAPTER

FIVE

LISCENSING

This work is in general licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. Part of this work is derived and copied from Zeek, Broker, and file-extraction all with **BSD 3-Clause License**, which shall be dual-licensed under the two licenses.

Original developed part of this software and associated documentation files (the "*Software*") are hereby licensed under the **Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License**. No permits are foreordained unless granted by the author and maintainer of the *Software*, i.e. Jarry Shaw.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

B

C

D

E

S

U

W