Domain Collector

A CLI tool used to gather domains and related information from various sources into a data set.

Authors: Adam Horák, Jan Polišenský, Radek Hranický
NES@FIT RESEARCH GROUP, FACULTY OF INFORMATION TECHNOLOGY, BRNO UNIVERSITY OF TECHNOLOGY, 2023

Domain Collector enables automated collection, aggregation and storage of knowledge about currently known trusted and dangerous domains on the Internet. It uses resources such as Cisco Umbrella and MISP Threat Sharing feeds, particularly VirusTotal, PhishTank, and OpenPhish, to obtain domain names. It then downloads and processes additional information about individual domains based on:

1. active interaction – server response to ICMP echo messages, open known ports,
2. external sources such as DNS, WHOIS/RDAP, information from TLS certificates, etc.

The acquired knowledge is then stored in a MongoDB database where it can be used for other purposes (rule creation for application firewalls or IDS/IPS systems, threat intelligence, machine learning, and cyber threat detection research).

It works in two phases:

- **Loading**: Collects categorized domain names from various sources and stores it in a mongo database.
- **Resolving**: Resolves select domain data and adds it to the database records.

The flow is shown in Figure 1. Both phases run independently. You can load directly from a file or use a source list to load from multiple sources. Domains are stored for future resolving and the data can be updated incrementally.

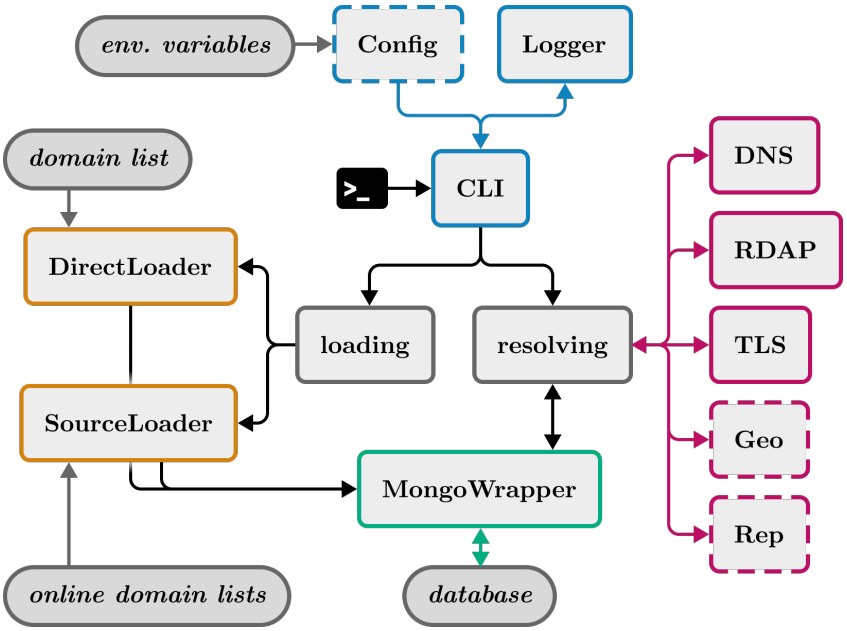

Figure 1: The architecture of Domain Collector

Requirements

- Python 3.9+
- MongoDB instance
- Installed pip modules from requirements.txt

To start, create a .env file in the root directory (or add to your environment directly) with the following variables:

```
DR_MONGO_URI=<mongo connection string>
DR_NERD_TOKEN=<nerd api token if you have one>
DR_MISP_KEY=<misp api key if you have one>
```

Without the connection string, it will default to an unauthenticated local instance at `mongodb://localhost:27017/`.

See **config.py** to change the database name and other defaults. This is also where you configure service URLs, such as NERD and your MISP instance.

Note that to use GeoIP, you need to download the GeoLite2 databases from MaxMind and place them in the **data/geolite** directory.

Usage

The interpreter invocation (ex. `python3`) will be omitted from the following examples. You can use `python` or `python3` depending on your system. Use python version 3.9 or later.

Note that commands that provide interactivity can be used with the `-y` or `--yes` flag to skip the interactive prompts and start straight away.

You can also get help at any point with the `--help` flag. Use it with commands to get information about the command and its arguments. Use it with the main script to get a list of available commands.

Loading

Start by loading domains into the database with the `load` or `load-misp` command. When loading domains, provide a label (such as *benign* or *malign*) to categorize the domains. This label will be used as the collection name in the database.

Loaded domain's records will have the **label**, source, timestamps, and a **category**, which is either automatically inferred from the source or MISP feed configuration or set to *unknown*. The category is meant to distinguish between different types of domains, such as *malware* or *phishing*.

The `load` command loads from a file or a list of sources:

```
main.py load [options] <file>

Options:
  -l, --label TEXT                Label for loaded domains
  -d, --direct                    Load directly from the file
```

You must provide a path to a file that will be read. If it's a source file (a list of domains to be loaded), use the *direct* flag to load directly from the file. Otherwise, it will be read as a list of sources to load from. Source lists are CSV files described in the *Source Lists* section.

Use the *label* option to set the label for the loaded domains. If you don't provide a label, it will default to *benign*. The label is also the collection name in the database. Category will be *unknown* if loading directly from a source file.

The `load-misp` command loads from a MISP instance:

```
main.py load-misp [options] <feed>

Options:
  -l, --label TEXT  Label for loaded domains
```

Provide the name of the MISP feed to load from. The feeds are configured in **config.py** in the *MISP_FEEDS* dictionary. The CLI will only allow feed names that are configured in the dictionary. You can also use the help flag to get a list of available choices.

The feeds dictionary defines the available options (feed names) as keys and the values are a tuple of the feed ID from MISP and the category the feed belongs to, such as *phishing*.

Use the *label* option to set the label for the loaded domains. If you don't provide a label, it will default to *misp*. The label is also the collection name in the database.

Resolving

HEADS UP: Pinging requires root privileges, because it constructs raw sockets. If you don't have root privileges, you can proceed as normal and the pings will be skipped, but errors will be logged.

To resolve domains in the database, use the `resolve` command:

```
main.py resolve [options]

Options:
  -t, --type [basic|geo|rep|ports]  Data to resolve
  -l, --label TEXT                  Label for loaded domains
  -e, --retry-evaluated             Retry resolving fields that have
failed before
  -n, --limit INTEGER               Limit number of domains to resolve
  -s, --sequential                  Resolve sequentially instead of in
parallel
```

Use the *type* option to specify the type of data to resolve. The default is *basic*. Resolving is separated into different types to allow for more granular control over the resolving process. The available types are:

- **basic**: Resolves most of the data available in the database. Domain data include DNS, RDAP, TLS certificates. For found IPs, it's RDAP and alive status (ping).
- **geo**: Resolves geolocation data for IPs.
- **rep**: Resolves reputation data for IPs (NERD only for now).

- **ports**: Performs port scan for IPs. This is slow and not recommended. Ports are currently hardcoded.

Use the *label* option to specify the label of the domains to resolve. If you don't provide a label, it will default to *benign*. The label is also the collection name in the database.

Use the *retry-evaluated* flag to retry resolving fields that have failed before. Failed means that the resolution for that field could not be completed for some reason deemed unrecoverable. This is useful if you want to retry resolving those.

Use the *limit* option to limit the number of domains to resolve. This is mostly used for debugging.

Use the *sequential* flag to resolve sequentially instead of in parallel worker threads. This is useful if you want to resolve via an API that has rate limits or any other constraints that could make parallel resolving problematic. Sequential resolving is much slower, however.

Stats

To get stats about the datasets, use the `stats` command:

```
main.py stats [options]

Options:
  -c, --collections              Collections to show stats for
  -w, --write                    Write stats to stats.json file
  -g, --geo                      Write coords to csv files instead
```

Use the *collections* option to specify the collections to show stats for. If you don't provide a list of collections, it will default to *misp* and *benign*. Specify multiple collections by repeating the option, such as `-c misp -c benign`.

Without the *write* flag, the stats will be printed to the console. Use the *write* flag to write the stats to a JSON file named *stats.json* in the root directory. Note that this is different from redirecting the output to a file. The JSON file will contain the stats in a JSON object, while redirecting the output will write the stats in a human-readable format.

The *geo* flag is used to get coordinates of all IPs in the collections and write them to CSV files. The CSV files will be named `coords_{collection}` after the collection they belong to. You can use these files to visualize the data on a map.

Dry Resolve

To try out resolving a single domain and print the results, use the `try` command:

```
main.py try [options] <domain>

Options:
  -p, --with-ports  Scan ports
```

Provide a domain name to try resolving. The *with-ports* flag will perform a port scan on the domain as well. Remember that pinging requires root privileges (see above).

Domain Data Format

Each domain name is stored in a document in the database. These follow a common loose schema, but not all fields are guaranteed to be present. You can see how the data is structured in the **datatypes.py** file.

In addition to the data fields, the document stores remarks and timestamps for when each field was resolved. This allows the program to skip resolving fields that have already been resolved or that failed for fatal reasons, such as a TLS certificate simply not existing. The logic to determine whether a field should be resolved or not goes like this:

- If the field is empty and the timestamp is empty, resolve the field. Either it has never been resolved or it failed for a recoverable reason.
- If the field is empty and the timestamp is recorded, it means that the field failed to resolve for some fatal reason. Resolve only if the *retry-evaluated* flag is set.
- In all other cases, the field has been resolved and the timestamp is recorded. Skip resolving the field.

In general, for each type of data, there is a nested object with the data. For example, the *dns* field contains the DNS data for the domain. The *ips* field contains a list of IP objects, each with its own fields.

Each document also stores timestamps for when the domain was sourced and when it was last resolved. The *source* field contains the source URI that the domain was sourced from. The *category* field contains the category of malicious activity that the domain is associated with.

Categories can be one of *unknown*, *malware*, *phishing*, *spam*, *ads*, *cryptomining*, *dga*.

Source Lists

Source lists are CSV files that contain a list of sources to load from. See the `data/Blacklists.csv` file for an example. There are several columns in the CSV file that can be used to specify the source URI and the category of malicious activity:

- **source**: The source URI. This is the only required column.
- **category**: The category of malicious activity that the domain is associated with. This is optional and defaults to *unknown*. Irrelevant for benign domains for example. See above for a list of available categories.
- **category source**: This determines the source of the category. If set to the value *this*, the category will be taken from the *category* column. Other values (*txt* and *csv*) are used to read the category from the source itself. This is useful if the source contains the category in its data and it differs for each domain.

There are two more columns that are used when the category source is set to *txt* or *csv*. These modes read the source as plain text or CSV respectively and use the category from the source itself. These two columns are used to specify how to read the category from the source. They are:

- **getter**: The getter specifies how to get the category from the source. - for plain text sources, the getter is a regular expression that is matched against the current line in the source. The first capture group in the regex is used as the category. - for CSV sources, the getter specifies the delimiter and column number to get the category from. The format is *delimiter* immediately followed by *column*. For example, `;2` will use semicolon as the delimiter and find the column with index 2.
- **mapper**: The mapper is used to map the category from the source to the categories used by the program. It's a list of mappings in the format *regex=category* separated by semicolons. The first mapping that matches the category from the source is used. If no mapping matches, the category is set to *unknown*.

Example for getters and mappers

Picture a plain text source that contains a list of domains and their categories in the format `domain - category`, for example:

```
example.com - Mallware
example2.com - Fishing
example3.com - Phish
```

Since there are multiple categories, we need to set the category source to *txt* and specify the getter and mapper. The getter will need to match the value after the dash and the mapper will need to map the category from the source to the categories used by the program, since the source list uses different naming. Thus, our source list will have the following for this source:

source	category	cat. source	getter	mapper
https://urls.org/domains.txt	-	txt	`-\s(.*)`	`Mallware=malware;.*=phishing`

The getter captures everything after the dash and whitespace. The mapper maps `Mallware` to `malware` and everything else to `phishing`. As you can see, this is fairly flexible and can be used to source various lists for the categories used by the program.

Using source lists with the source loader

The source loader takes source files from the list and can read plain text files, CSV files and JSON files. It can also unzip archives containing these files. The format of the source file is determined by the file extension. If the file extension is not recognized, the loader will try to read it as a plain text file.

To use a source list, simply provide the path to the source list file as the source URI. The source loader will then read the source list and load domains from the sources in it. Specifying indices for the above mentioned columns is not yet implemented, and the columns are hardcoded for use with the default source list. This is priority for the next update.

Acknowledgements

This software is supported by Smart information technology for a resilient society project, no. FIT-S-23-8209 granted by Brno University of Technology.

We also thank other researchers, namely Ondřej Ryšavý, Petr Matoušek, Ondrej Lichtner, Kamil Jeřábek and Jan Pluskal who provided us with assistance, useful tips and feedback during the creation of the software.