Cloudmarker Documentation

Release 0.1.0

Cloudmarker Authors and Contributors

May 20, 2020

Contents

Cloudmarker is a cloud monitoring tool and framework.

CHAPTER 1

Contents

Table of Contents:

CHAPTER 2

What is Cloudmarker?

Cloudmarker is a cloud monitoring tool and framework. It can be used as a ready-made tool that audits your Azure or GCP cloud environments as well as a framework that allows you to develop your own cloud monitoring software to audit your clouds.

As a monitoring tool, it performs the following actions:

- Retrieves data about each configured cloud using the cloud APIs.
- Saves or indexes the retrieved data into each configured storage system or indexing engine.
- Analyzes the data for potential issues and generates events that represent the detected issues.
- Saves the events to configured storage or indexing engines as well as sends the events as alerts to alerting destinations.

Each of the above four aspects of the tool can be configured via a configuration file.

For example, the tool can be configured to pull data from Azure and index its data in Elasticsearch while it also pulls data from GCP and indexes the GCP data in MongoDB. Similarly, it is possible to configure the tool to check for unencrypted disks in Azure, generate events for it, and send them as alerts by email while it checks for insecure firewall rules in both Azure and GCP, generate events for them, and save those events in MongoDB.

This degree of flexibility to configure audits for different clouds in different ways comes from the fact that Cloudmarker is designed as a combination of lightweight framework and a bunch of plugins that do the heavylifting for retrieving cloud data, storing the data, analyzing the data, generating events, and sending alerts. These four types of plugins are formally known as cloud plugins, store plugins, event plugins, and alert plugins, respectively.

As a result of this plugin-based architecture, Cloudmarker can also be used as a framework to develop your own plugins that extend its capabilities by adding support for new types of clouds or data sources, storage or indexing engines, event generation, and alerting destinations.

CHAPTER 3

Why Cloudmarker?

One might wonder why we need a new project like this when similar projects exist. When we began working on this project in 2017, we were aware of similar tools that supported AWS and GCP but none that supported Azure at that time. As a result, we wrote our own tool to support Azure. We later added support for GCP as well. What began as a tiny proof of concept gradually turned into a fair amount of code, so we thought, we might as well share this project online, so that others could use it and see if they find value in it.

So far, some of the highlights of this project are:

- It is simple. It is easy to understand how to use the four types of plugins (clouds, stores, events, and alerts) to perform an audit.
- It is excellent at creating an inventory of the cloud environment.
- The data inventory it creates is easy to query.
- It is good at detecting insecure firewall rules and unencrypted disks. New detection mechanisms are coming up.

We also realize that we can add a lot more functionality to this project to make it more powerful too. See the *Wishlist* section below to see new features we would like to see in this project. Our project is hosted on GitHub at https://github.com/cloudmarker/cloudmarker. Contributions and pull requests are welcome.

We hope that you would give this project a shot, see if it addresses your needs, and provide us some feedback by posting a comment in our feedback thread or by creating a new issue.

CHAPTER 4

Features

Since Cloudmarker is not just a tool but also a framework, a lot of its functionality can be extended by writing plugins. However, Cloudmarker also comes bundled with a default set of plugins that can be used as is without writing a single line of code. Here is a brief overview of the features that come bundled with Cloudmarker:

- Perform scheduled or ad hoc audits of cloud environment.
- Retrieve data from Azure and GCP.
- Store or index retrieved data in Elasticsearch, MongoDB, Splunk, and the file system.
- Look for insecure firewall rules and generate firewall rule events.
- Look for unencrypted disks (Azure only) and generate events.
- Send alerts for events via email and Slack as well as save alerts in one of the supported storage or indexing engines (see the third point above).
- Normalize firewall rules from Azure and GCP which are in different formats to a common object model (`"com"`) so that a single query or event rule can search for or detect issues in firewall rules from both clouds.

CHAPTER 5

Wishlist

- Add more event plugins to detect different types of insecure configuration.
- Normalize other types of data into a common object model (`"com"`) just like we do right now for firewall rules.

CHAPTER 6

Install

Perform the following steps to set up Cloudmarker.

1. Create a virtual Python environment and install Cloudmarker in it:

```
python3 -m venv venv
. venv/bin/activate
pip3 install cloudmarker
```

2. Run sanity test:

```
cloudmarker -n
```

   The above command runs a mock audit with mock plugins that generate some mock data. The mock data generated can be found at `/tmp/cloudmarker/`. Logs from the tool are written to the standard output as well as to `/tmp/cloudmarker.log`.

   The `-n` or `--now` option tells Cloudmarker to run right now instead of waiting for a scheduled run.

To learn how to configure and use Cloudmarker with Azure or GCP clouds, see Cloudmarker Tutorial.

CHAPTER 7

Develop

This section describes how to set up a development environment for Cloudmarker. This section is useful for those who would like to contribute to Cloudmarker or run Cloudmarker directly from its source.

1. We use primarily three tools to perform development on this project: Python 3, Git, and Make. Your system may already have these tools. But if not, here are some brief instructions on how they can be installed.

   On macOS, if you have Homebrew installed, then these tools can be be installed easily with the following command:

   ```
   brew install python git
   ```

   On a Debian GNU/Linux system or in another Debian-based Linux distribution, they can be installed with the following commands:

   ```
   apt-get update
   apt-get install python3 python3-venv git make
   ```

   On a CentOS Linux distribution, they can be installed with these commands:

   ```
   yum install centos-release-scl
   yum install git make rh-python36
   scl enable rh-python36 bash
   ```

   Note: The `scl enable` command starts a new shell for you to use Python 3.

   On any other system, we hope you can figure out how to install these tools yourself.

2. Clone the project repository and enter its top-level directory:

   ```
   git clone https://github.com/cloudmarker/cloudmarker.git
   cd cloudmarker
   ```

3. Create a virtual Python environment for development purpose:

   ```
   make venv deps
   ```

This creates a virtual Python environment at `~/.venv/cloudmarker`. Additionally, it also creates a convenience script named `venv` in the current directory to easily activate the virtual Python environment which we will soon see in the next point.

To undo this step at anytime in future, i.e., delete the virtual Python environment directory, either enter `rm -rf venv ~/.venv/cloudmarker` or enter `make rmvenv`.

4. Activate the virtual Python environment:

```
. ./venv
```

5. In the top-level directory of the project, enter this command:

```
python3 -m cloudmarker -n
```

This generates mock data at `/tmp/cloudmarker`. This step serves as a sanity check that ensures that the development environment is correctly set up and that the Cloudmarker audit framework is running properly.

6. Now that the project is set up correctly, you can create a `cloudmarker.yaml` to configure Cloudmarker to scan/audit your cloud or you can perform more development on the Cloudmarker source code. See Cloudmarker Tutorial for more details.

7. If you have set up a development environment to perform more development on Cloudmarker, please consider sending a pull request to us if you think your development work would be useful to the community.

8. Before sending a pull request, please run the unit tests, code coverage, linters, and document generator to ensure that no existing test has been broken and the pull request adheres to our coding conventions:

```
make test
make coverage
make lint
make docs
```

To run these four targets in one shot, enter this "shortcut" target:

```
make checks
```

Open `htmlcov/index.html` with a web browser to view the code coverage report.

Open `docs/_build/html/index.html` with a web browser to view the generated documentation.

CHAPTER 8

Resources

Here is a list of useful links about this project:

- Documentation on Read The Docs
- Latest release on PyPI
- Source code on GitHub
- Issue tracker on GitHub
- Changelog on GitHub
- Cloudmarker channel on Slack
- Invitation to Cloudmarker channel on Slack

CHAPTER 9

Support

To report bugs, suggest improvements, or ask questions, please create a new issue at http://github.com/cloudmarker/cloudmarker/issues.

CHAPTER 10

License

This is free software. You are permitted to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of it, under the terms of the MIT License. See LICENSE.rst for the complete license.

This software is provided WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See LICENSE.rst for the complete disclaimer.

CHAPTER 11

Tutorial

11.1 Cloudmarker Tutorial

Cloudmarker is a cloud monitoring tool and framework.

11.1.1 Install

1. Create a virtual Python environment and install Cloudmarker in it:

```
python3 -m venv venv
. venv/bin/activate
pip3 install cloudmarker
```

2. Run sanity test:

```
cloudmarker -n
```

   The above command runs a mock audit with mock plugins that generate some mock data. The mock data generated can be found at `/tmp/cloudmarker/`. Logs from the tool are written to the standard output as well as to `/tmp/cloudmarker.log`.

   The `-n` or `--now` option tells Cloudmarker to run right now instead of waiting for a scheduled run.

11.1.2 Get Started

Cloudmarker's behaviour is driven by configuration files written in YAML format. Cloudmarker comes with two built-in mock plugins known as `MockCloud` and `MockEvent`. These mock plugins are useful in generating some mock data to test out the Cloudmarker framework and familiarize oneself with how Cloudmarker can be configured.

We will first see how to configure the `MockCloud` plugin, just so that we can quickly get started with understanding the configuration file format without having to work out how to provide Cloudmarker access to real clouds. We will see how to work with real clouds later in this document. Follow these steps to get started:

1. Create a config file named `cloudmarker.yaml` in the current directory with the following content:

```
plugins:
  mymockcloud:
    plugin: cloudmarker.clouds.mockcloud.MockCloud
    params:
      record_count: 5
      record_types:
        - apple
        - ball
        - cat

audits:
  mymockaudit:
    clouds:
      - mymockcloud
    stores:
      - filestore
    events:
      - mockevent
    alerts:
      - filestore

run:
  - mymockaudit
```

2. Enter this command to run Cloudmarker:

```
cloudmarker -n
```

3. Examine the output in `/tmp/cloudmarker/mymockaudit_mymockcloud.json`. It should contain a JSON array with 5 objects as defined in the `record_count` value in the config. There are `record_type` fields in these objects that cycle between the values `"apple"`, `"ball"`, and `"cat"` as defined in the `record_types` value in the config. The data we see in this output file is generated by the *`cloudmarker.clouds.mockcloud.MockCloud`* plugin defined under the `mymockcloud` config key.

4. Now examine the output in `/tmp/cloudmarker/mymockaudit_mockevent`. It should contain a JSON array with 2 objects. This data is generated by the `cloudmarker.clouds.mockcloud.MockEvent` plugin referred to as `mockevent`. The `mockevent` config key is defined in the *built-in base config*.

5. Note that the mock audit files written at `/tmp/cloudmarker/` have names that are composed of audit key name, underscore, and plugin key name. These files are written by the `cloudmarker.clouds.filestore.FileStore` plugin which is specified in the config as `filestore`. The `filestore` config key is defined in the *built-in base config*.

Configuration Format

Let us take a closer look at the config file format in the previous section:

```
plugins:
  mymockcloud:
    plugin: cloudmarker.clouds.mockcloud.MockCloud
    params:
      record_count: 5
      record_types:
        - apple
        - ball
```

(continues on next page)

(continued from previous page)

```
        - cat

audits:
  mymockaudit:
    clouds:
      - mymockcloud
    stores:
      - filestore
    events:
      - mockevent
    alerts:
      - filestore

run:
  - mymockaudit
```

There are three top-level config keys: `plugins`, `audits`, and `run`. These top-level keys and their values are input to the Cloudmarker framework. They tell the framework what to do. Let us see each top-level key in more detail.

plugins

The `plugins` key defines one or more plugin configs. In the above example, we have defined only one plugin config for the *`cloudmarker.clouds.mockcloud.MockCloud`* plugin. A plugin is a Python class that implements a few methods required by the Cloudmarker framework. In this case, the `MockCloud` plugin has the code to generate some mock data for the purpose of testing other plugins.

Under the `plugins` key, we have one or more user-defined keys that name our plugin configs. In this example, we have defined one plugin config and chosen that name `mymockcloud` for it. We could name this anything. This name appears in the logs, so it is good to name this meaningfully.

Under the user-defined key for a plugin, there are at most two keys:

- `plugin`: Its value is the fully qualified class name of the plugin class.
- `params`: Its value is a mapping of key-value pairs that specify the keyword arguments to pass to the plugin class constructor expression. For example, see the API documentation of `MockCloud` by clicking on this link: *`cloudmarker.clouds.mockcloud.MockCloud`*. We can see that the config key names `record_count` and `record_types` under the the `params` config key match the parameter names of the `MockCloud` plugin.

audits

The `audits` key defines one or more audit configs. In the above example, we have defined only one audit config to generate mock data using the plugin defined under the `mymockcloud` config key earlier.

Under the `audits` key, we have one or more user-defined keys that name our audit configs. In this example, we have defined one audit config named key `mymockaudit`. This name appears in the logs, so it is good to name this meaningfully.

Under the user-defined key for an audit, there are four keys:

- `clouds`: Its value is a list of config keys defined under the `plugins` key. Each key should refer to a cloud plugin.
- `stores`: Its value is a list of config keys defined under the `plugins` key. Each key should refer to a store plugin.

- `events`: Its value is a list of config keys defined under the `plugins` key. Each key should refer to an event plugin.
- `alerts`: Its value is a list of config keys defined under the `plugins` key. Each key should refer to a store or alert plugin.

The Cloudmarker framework instantiates all the plugins in an audit and then lets the cloud plugins generate cloud records. Within an audit, records generated by each cloud plugin are then fed to each store and event plugin configured in the same audit.

Each cloud plugin generates data, typically, by connecting to a cloud and pulling cloud data related to resources configured in the cloud. Each cloud plugin emits this data in Python dictionary formats (appears as JSON when written to files or a storage/indexing system that can store JSON documents). We call each such dictionary object (or JSON document) as a *record*.

Each event plugin then receives the data generated by the cloud plugins configured within the same audit. An event plugin checks each record for security issues or some pattern. If a security issue is found in some record or if the pattern being checked for is found, then the event plugins generate one or more events for it. These events are fed to each alert plugin in the same audit.

Each store plugin takes the data fed to it and sends it to the store destination. A store destination is typically a storage or indexing engine such as Elasticsearch, Splunk, etc.

Each alert plugin takes events fed to it and sends the events to an alerting destination. A store plugin can also function as an alert plugin and vice-versa. From the framework's perspective, there is no difference between store and alert plugin classes because they implemented the same methods. The only difference is that the store plugins are mentioned under the `stores` key in an audit config and the alert plugins are mentioned under the `alerts` key in an audit config. However, some alert plugins such as the ones to send events as email alerts or Slack messages make sense only as alert plugins and not as store plugins. That's because we wouldn't want to email the entire cloud data to email recipients or Slack users but we might want to email just the events as notifications to email recipients or Slack users.

run

Finally, the `run` key defines the audits we want to run. Its value is a list of one or more user-defined audit keys.

Base Configuration

In the above examples, we defined `mymockcloud` under the `plugins` key but we did not define `filestore` or `mockevent` although we used them under the `audits` key. That's because they are already defined in the built-in base config. Enter this command to see the built-in base config:

```
cloudmarker --print-base-config
```

Alternatively, see the complete built-in base config here: *`cloudmarker.baseconfig`*.

The config in `cloudmarker.yaml` or any other user-specified config files is merged with the built-in base config to arrive at the final working config. The merging rules are described in the next section.

Cascading Configuration

By default, Cloudmarker looks for the following config files in the order specified:

- `/etc/cloudmarker.yaml`
- `~/.cloudmarker.yaml`
- `~/cloudmarker.yaml`

- `cloudmarker.yaml`

Note that the *built-in base config* is always used. If a config file in the list above is missing, it is ignored. If all config files in the list above are missing, then only the built-in base config is used.

If one or more config files are present, they are merged together with the built-in base config to arrive at the final working config. The built-in base config is loaded first. Then the config files are loaded and merged in the order specified in the list above. A config that is loaded later has higher precedence in case of conflicting values for the same key.

A custom list of config files to look for can be specified with the `-c` or `--config` option. For example, the following command first loads the built-in base config, then `foo.yaml` from the current directory, and then `bar.yaml` from the current directory:

```
cloudmarker -n -c foo.yaml bar.yaml
```

It means that in case of conflicting values for the key, the builtin-base config has the lowest priority and `bar.yaml` has the highest priority.

Here is a precise specification of how two configs are merged:

- If a key in the first config does not exist in the second config, then the final config contains this key with its value intact.
- If a key in the second config does not exist in the first config, then the final config contains this key with its value intact.
- If a key in the first config also exists in the second config, then the final config contains this key and its value is the value found in the second config.

This means, if a key with a list value exists in the first config and the same key with another list value exists in the second config, then the final config is the key with the list value in the second config. The final config *does not* contain the key with both lists merged together as its value.

For example, let us assume that `foo.yaml` contains this key:

```
run:
  - mockaudit
  - fooaudit
```

And `bar.yaml` contains this key:

```
run:
  - baraudit
```

Then `cloudmarker -n -c foo.yaml bar.yaml` leads to the following value for this config key:

```
run:
  - baraudit
```

Note how the list value of `bar.yaml` replaced the list value of `foo.yaml` while merging. In other words, when we talk about merging of configs, only keys are merged, i.e., keys from both configs are picked for the final working config. Values are not merged.

When there are multiple config files to be merged, the first config file is merged with the base config file, then the second config file is merged with the result of the previous merge, and so on.

11.1.3 Cloud Plugins

AzCloud

To get started with a real audit, it is necessary to configure Cloudmarker with an actual cloud such as Azure or GCP. In this section, we see how to configure Cloudmarker for Azure with the `AzCloud` plugin.

This plugin is offered by the *`cloudmarker.clouds.azcloud.AzCloud`* plugin class.

Perform the following steps to configure this plugin:

1. At first follow this how-to document at https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-create-service-principal-portal to register an application in Azure Active Directory to allow Cloudmarker to access your Azure resources.
2. After completing the above step, create a config file named `cloudmarker.yaml` in the current directory with this content:

```
plugins:
  myazcloud:
    plugin: cloudmarker.clouds.azcloud.AzCloud
    params:
      tenant: null
      client: null
      secret: null

audits:
  myazaudit:
    clouds:
      - myazcloud
    stores:
      - filestore
    events:
      - firewallruleevent
    alerts:
      - filestore

run:
  - myazaudit
```

3. Then replace the `null` values for `tenant`, `client`, and `secret` as described below:
   - `tenant`: This is the tenant ID obtained from following the "Get tenant ID" section of the how-to document. This is also known as the directory ID. To find this value, go to Azure Portal > Azure Active Directory > Properties > Directory ID. This value is also available in the newly created application at Azure Portal > Azure Active Directory > App Registrations > (the app) > Directory (tenant) ID.
   - `client`: This is the application ID created in the "Get application ID and authentication key" section of the how-to document. This value is also available at Azure Portal > Azure Active Directory > App Registrations > (the app) > Application (client) ID.
   - `secret`: This is the secret password created in the "Get application ID and authentication key" section of the how-to document. This valuable is available only while creating a new secret at Azure Portal > Azure Active Directory > App Registrations > (the app) > New client secret.
4. After setting these values, enter this command to run Cloudmarker:

```
cloudmarker -n
```

5. After Cloudmarker completes running, check these files in `/tmp/cloudmarker/`:
   - `myazaudit_myazcloud.json`: This file contains the data obtained from Azure cloud by the *`cloudmarker.clouds.azcloud.AzCloud`* plugin configured under the `myazaudit` config key.

- `myazaudit_firewallruleevent.json`: This file contains insecure firewall rules detected by the *`cloudmarker.events.firewallruleevent.FirewallRuleEvent`* referred to with the key name `firewallruleevent` in the config. Note that `firewallruleevent` config key is defined in the *built-in base config*.

AzVM

This is another plugin for Azure that has a narrower but deeper scope than the `AzCloud` plugin described in the previous section. It pulls only virtual machine (VM) data with more details about each VM.

This plugin is offered by the *`cloudmarker.clouds.azvm.AzVM`* plugin class.

Perform the following steps to use the `AzVM` plugin:

1. As mentioned in the previous section, register an application in Azure Active Directory to allow Cloudmarker to access your Azure resources.
2. After completing the above step, create a config file named `cloudmarker.yaml` in the current directory with this content:

```
plugins:
  myazvm:
    plugin: cloudmarker.clouds.azvm.AzVM
    params:
      tenant: f3cfe067-d008-48f3-b026-cf0dd7409b25
      client: 6c4980e2-2652-466d-8157-853f9d0a288f
      secret: 4FAU+gYAkl96zbnlXZqu25d5iZBlDhzj0EHD8fi6HR8=

audits:
  myazaudit:
    clouds:
      - myazvm
    stores:
      - filestore
    events:
      - firewallruleevent
      - azvmosdiskencryptionevent
      - azvmdatadiskencryptionevent
    alerts:
      - filestore

run:
  - myazaudit
```

3. Then replace the `null` values for `tenant`, `client`, and `secret` with actual values as described in the previous section.
4. After setting these values, enter this command to run Cloudmarker:

```
cloudmarker -n
```

5. After Cloudmarker completes running, check the generated files in `/tmp/cloudmarker/`.

Let us discuss how `AzCloud` and `AzVM` are different.

`AzCloud` pulls data at subscription level. It first connects to Azure with the specified credentials, then queries for all subscriptions it has access to, and then loops over each subscription and makes one API call per subscription per resource type to pull all resources of that type. It pulls data related to virtual machines (VMs), application gateways, load balancers, network interface controllers (NICs), network security groups (NSGs), etc.

`AzVM` on the other hand pulls data at the virtual machine level. It makes one API call per VM. Thus, it makes more number of API calls. It retrieves only VM data but it gets more detailed VM data. For example, this plugin also obtains the power state, the disk encryption status, etc. of the VM. These detailed level of information cannot be obtained by the `AzCloud` plugin.

To understand the difference between the two plugins better, consider an environment where there are 5 subscriptions such that each subscription has exactly 20 VMs and 50 NSGs. So, there are a total of 100 VMs and 250 NSGs. `AzCloud` would make only 5 API calls to pull data for all 100 VMs and another 5 API calls to pull data for all NSGs. On the other hand, `AzVM` would make 100 API calls to pull data for all VMs. `AzVM` cannot pull data for NSGs or any other type of resources. However, the VM data obtained by `AzVM` contains power state, disk encryption status, and other detailed information. `AzCloud` data does not pull such detailed information.

In general, `AzCloud` runs faster due to less number of API calls and is usually sufficient for most types of cloud monitoring use cases. `AzVM` is necessary only for advanced use cases such as monitoring whether a particular VM is running or stopped, if its disks are encrypted or not, etc.

Note in the config above that event plugins *cloudmarker.events.azvmosdiskencryptionevent.AzVMOSDiskEncryptionEvent* and *cloudmarker.events.azvmdatadiskencryptionevent.AzVMDataDiskEncryptionEvent* referred to with the built-in base config keys `azvmosdiskencryptionevent` and `azvmdatadiskencryptionevent` can be used with `AzVM`. These plugins work only with `AzVM` records and generates events if OS disks and data disks are found. They ignore records generated by any other cloud plugins.

GCPCloud

Follow these steps to get started with auditing a GCP cloud environment.

1. Follow the steps at https://cloud.google.com/iam/docs/creating-managing-service-account-keys to create a service account key using the GCP Console and download it as a file named `keyfile.json`.
2. Then create a config file name `cloudmarker.yaml` with this content:

```
plugins:
  mygcpcloud:
    plugin: cloudmarker.clouds.gcpcloud.GCPCloud
    params:
      key_file_path: keyfile.json
      zone: null

audits:
  mygcpaudit:
    clouds:
      - mygcpcloud
    stores:
      - filestore
    events:
      - firewallruleevent
    alerts:
      - filestore

run:
  - mygcpaudit
```

3. Replace the value of `zone` key to the zone name in which your resources reside. The zone name can be found in GCP Console > (select project) > Go to Compute Engine. An example of zone name is `us-east1-b`. Note: We are aware that the requirement of providing a specific zone name in the config makes this plugin less flexible. This will be fixed in the next release. The fix would allow the plugin to discover resources in all zones automatically.

4. Now enter this command to run Cloudmarker:

```
cloudmarker -n
```

5. Now examine these files generated by Cloudmarker at `/tmp/cloudmarker/`:

   - `mygcpaudit_mygcpcloud.json`: This file contains the data obtained from GCP cloud by the *`cloudmarker.clouds.gcpcloud.GCPCloud`* plugin configured under the `mygcpaudit` config key.
   - `mygcpaudit_firewallruleevent.json`: This file contains insecure firewall rules detected by the *`cloudmarker.events.firewallruleevent.FirewallRuleEvent`* plugin referred to with the key name `firewallruleevent` in the *built-in base config*.

MockCloud

The `MockCloud` plugin has already been discussed in the *Get Started* section once. A config key named `mockcloud` already configures this plugin in the *built-in base config* as follows:

```
plugins:
  mockcloud:
    plugin: cloudmarker.clouds.mockcloud.MockCloud
```

There are no parameters specified for this plugin in the built-in base config because this plugin class already has default keyword parameters. See *`cloudmarker.clouds.mockcloud.MockCloud`* for the keyword parameters with default values. By default, it generates 10 mock records such that `record['ext']['record_type']` alternate between `'foo'` and `'bar'` where `record` represents each JSON object generated by this plugin.

To override the default behaviour to, say, generate 20 records with record types that alternate between *'foo'*, *'bar'*, and *'baz'*, we could override the `mockcloud` config key defined in the built-in base config. To do so, create a file named `cloudmarker.yaml` with the following content only:

```
plugins:
  mockcloud:
    params:
      record_count: 20
      record_types:
        - foo
        - bar
        - baz
```

Then run Cloudmarker with this command:

```
cloudmarker -n
```

Note that we did not specify the `plugin` key under `mockcloud` here because that is already available in the base config (see *`cloudmarker.baseconfig`*). Similarly, we did not define `audits` or `run` config keys here because they are also defined in the base config. We only defined what we needed to override in the base config.

11.1.4 Event Plugins

The event plugins have been discussed in the sections for cloud plugins above. Here is how the config keys for these plugins have been defined in the base config (see *`cloudmarker.baseconfig`*):

```
plugins:
  ...

  firewallruleevent:
    plugin: cloudmarker.events.firewallruleevent.FirewallRuleEvent

  azvmosdiskencryptionevent:
    plugin: cloudmarker.events.azvmosdiskencryptionevent.AzVMOSDiskEncryptionEvent

  azvmdatadiskencryptionevent:
    plugin: cloudmarker.events.azvmdatadiskencryptionevent.AzVMDataDiskEncryptionEvent
```

The ellipsis (`...`) in this example denote content omitted in the above example for the sake of brevity.

FirewallRuleEvent

The `FirewallRuleEvent` plugin can be used with both `AzCloud` and `GCPCloud` plugins. It looks for firewall rules that expose sensitive ports to the entire Internet and generates events for them.

This plugin is offered by the *cloudmarker.events.firewallruleevent.FirewallRuleEvent* plugin class.

By default, it monitors for insecure exposure of a fixed set of TCP ports. If that's okay for you, there is no need to define this plugin explicitly in the config file. The built-in base config key `firewallruleevent` can be used as is. However, if there is a need for monitoring a custom set of ports, then it can be overridden. Here is an example configuration that monitors for insecure exposure of ports 22 and 3389 in Azure cloud:

```
plugins:
  myazcloud:
    plugin: cloudmarker.clouds.azcloud.AzCloud
    params:
      tenant: null
      client: null
      secret: null

  firewallruleevent:
    params:
      ports:
        - 22
        - 3389

audits:
  myazaudit:
    clouds:
      - myazcloud
    stores:
      - filestore
    events:
      - firewallruleevent
    alerts:
      - filestore

run:
  - myazaudit
```

Remember to replace the `null` values in the config above with actual values before using this config.

AzVMOSDiskEncryptionEvent

The `AzVMOSDiskEncryptionEvent` plugin can be used with `AzVM` plugin. It looks for unencrypted OS disks attached to Azure virtual machines.

This plugin is offered by the *`cloudmarker.events.azvmosdiskencryptionevent.AzVMOSDiskEncryptionEvent`* plugin class.

An example usage of this plugin is available in the *AzVM* section. Since it only checks whether disks are encrypted or not (a binary decision), it does not accept any parameters that can be configured in config file. Therefore, it is recommended to use the built-in base config key named `azosdiskencryptionevent` for this plugin.

AzVMDataDiskEncryptionEvent

The `AzVMDataDiskEncryptionEvent` plugin can be used with `AzVM` plugin. It looks for unencrypted data disks attached to Azure virtual machines.

This plugin is offered by the *`cloudmarker.events.azvmdatadiskencryptionevent.AzVMDataDiskEncryptionEvent`* plugin class.

An example usage of this plugin is available in the *AzVM* section. Since it only checks whether disks are encrypted or not (a binary decision), it does not accept any parameters that can be configured in config file. Therefore, it is recommended to use the built-in base config key named `azdatadiskencryptionevent` for this plugin.

MockEvent

The `MockEvent` plugin can be used with `MockCloud` plugin. The `MockCloud` plugin generates data such that `record['raw']['data']` has an integer value that increments in each record where `record` here represents each record generated by `MockCloud`. The `MockEvent` plugin when used checks the value of `record['raw']['data']` in each input `record` and generates an event if this value is a multiple of some number (`3` by default).

This plugin is offered by the *`cloudmarker.events.mockevent.MockEvent`* plugin class.

We use `MockCloud` and `MockEvent` plugins together to test out the store and alert plugins.

In case, we want the `MockEvent` plugin to look for a multiple of some other number, say, `5`, we can override the built-in base config as follows:

```
plugins:
  mockevent:
    params:
      n: 5
```

11.1.5 Store Plugins

FileStore

We have been using the `FileStore` plugin already in the examples above. This plugin is good for quick testing because we can see the cloud data records and events written locally to a file that we can easily inspect.

This plugin is offered by the *`cloudmarker.stores.filestore.FileStore`* plugin class.

By default, it writes the output files to the `/tmp/cloudmarker/` directory. Here is how it can be configured to write the output files to another directory, say, `~/cloudmarker`:

```
plugins:
  filestore:
    params:
      path: ~/cloudmarker
```

On running Cloudmarker with this config, we would see that the output files have been written to `~/cloudmarker`, i.e., `$HOME/cloudmarker` or in other words, the `cloudmarker` directory under the home directory. Yes, `FileStore` performs tilde expansion to expand a path beginning with `~` to a user's home directory as mentioned here: `os.path.expanduser()`.

EsStore

The `EsStore` plugin can be used to send cloud data as well as events to an Elasticsearch cluster.

This plugin is offered by the *`cloudmarker.stores.esstore.EsStore`* plugin class.

In this section, we will use a Docker image of Elasticsearch to quickly get started with configuring this plugin. Here are the steps to set up a Docker container for Elasticsearch:

1. Enter the following command to run a Docker container with Elasticsearch instance:

   ```
   docker run -p 9200:9200 -p 9300:9300 \
              -e 'discovery.type=single-node' \
              docker.elastic.co/elasticsearch/elasticsearch:7.0.1
   ```

2. Ensure that Elasticsearch is able to index documents:

   ```
   curl -H 'Content-Type: application/json' \
        -X PUT http://localhost:9200/foo/foo/1?pretty \
        -d '{"a": "apple", "b": "ball"}'
   ```

3. Double-check that the document was indexed:

   ```
   curl http://localhost:9200/foo/_search?pretty
   ```

Now that Elasticsearch is running in a Docker container and indexing data, configure Cloudmarker to send data and events to it with the following steps:

1. Create `cloudmarker.yaml` with the following content to configure Cloudmarker to send mock cloud records and mock events to Elasticsearch:

   ```
   audits:
     mockaudit:
       stores:
         - filestore
         - esstore
       alerts:
         - filestore
         - esstore
   ```

   The above example is a very minimal config. It works because the `esstore` plugin config key is defined in the built-in base config and it sends data to a locally running Elasticsearch by default. Here is what a more elaborate config would look like:

   ```
   plugins:
     esstore:
       host: localhost
   ```

(continues on next page)

(continued from previous page)

```
    port: 9200
    index: cloudmarker

audits:
  mockaudit:
    stores:
      - filestore
      - esstore
    alerts:
      - filestore
      - esstore
```

2. Run Cloudmarker:

```
cloudmarker -n
```

3. Confirm that mock cloud records and mock events are indexed in Elasticsearch:

```
curl http://localhost:9200/cloudmarker/_search?pretty
```

MongoDBStore

The `MongoDBStore` plugin can be used to send cloud data as well as events to a MongoDB collection.

This plugin is offered by the *`cloudmarker.stores.mongodbstore.MongoDBStore`* plugin class.

In this section, we will use a Docker image of MongoDB to quickly get started with configuring this plugin. Here are the steps to set up a Docker container for MongoDB:

1. Enter the following commands to run a Docker container with MongoDB instance:

```
docker rm mongo; docker run --name mongo -p 27017:27017 mongo
```

2. Ensure that we can insert data into MongoDB:

```
docker exec -it mongo mongo foo --eval 'db.bar.insert({"a": "apple"})'
```

3. Double-check that the data was inserted:

```
docker exec -it mongo mongo foo --eval 'db.bar.find()'
```

Now that MongoDB is running in a Docker container, configure Cloudmarker to send data to it with these steps:

1. Create `cloudmarker.yaml` with the following content to configure Cloudmarker to send mock cloud records and mock events to MongoDB:

```
audits:
  mockaudit:
    stores:
      - filestore
      - mongodbstore
    alerts:
      - filestore
      - mongodbstore
```

The above example is a very minimal config. It works because the `mongodbstore` plugin config key is defined in the built-in base config and it sends data to a locally running MongoDB by default. Here is what a more elaborate config would look like:

```
plugins:
  mongodbstore:
    host: localhost
    port: 27017
    db: cloudmarker
    collection: cloudmarker
    username: null
    password: null

audits:
  mockaudit:
    stores:
      - filestore
      - mongodbstore
    alerts:
      - filestore
      - mongodbstore
```

If the MongoDB instance requires user authentication, then the `username` and `password` config keys should be set to the appropriate values.

2. Run Cloudmarker:

```
cloudmarker -n
```

3. Confirm that mock cloud records and mock events are indexed in Elasticsearch:

```
docker exec -it mongo mongo cloudmarker --eval 'db.cloudmarker.find()'
```

SplunkHECStore

The `SplunkHECStore` plugin can be used to send cloud data as well as events to a Splunk HTTP Event Collector (HEC).

This plugin is offered by the *`cloudmarker.stores.splunkhecstore.SplunkHECStore`* plugin class.

In this section, we will use a Docker image of Splunk to quickly get started with configuring this plugin. Here are the steps to set up a Docker container for Splunk:

1. Enter the following command to run a Docker container with Splunk instance with HTTP Event Collector (HEC):

```
docker run -e 'SPLUNK_START_ARGS=--accept-license' \
           -e 'SPLUNK_PASSWORD=admin123' \
           -e 'SPLUNK_HEC_TOKEN=token123' \
           -p 8000:8000 -p 8088:8088 splunk/splunk
```

2. Ensure that Splunk HEC is able to receive events:

```
curl -k https://localhost:8088/services/collector/event \
     -H 'Authorization: Splunk token123' \
     -d '{"event": "hello, world"}'
```

3. To double-check that Splunk received the event, visit http://localhost:8000/ with a web browser.

4. Then log into Splunk with username as `admin` and password as the password specified in the `docker` command in step 1 above.
5. Click on "Search & Reporting" on the left sidebar.
6. In the search box, enter * (asterisk) and click the search button. One event with the string `hello, world` should appear in the result.

Now that Splunk is running in a Docker container and accepting events via HEC, configure Cloudmarker to send data and events to it with the following steps:

1. Create `cloudmarker.yaml` with the following content to configure Cloudmarker to send mock cloud records and mock events to Splunk:

```
plugins:
  splunkstore:
    plugin: cloudmarker.stores.splunkhecstore.SplunkHECStore
    params:
      uri: https://localhost:8088/services/collector
      token: token123
      index: main
      ca_cert: false

audits:
  mockaudit:
    stores:
      - filestore
      - splunkstore
```

2. Run Cloudmarker with this configuration:

```
cloudmarker -n
```

3. Visit http://localhost:8000/ with a web browser.
4. Then log into Splunk with username as `admin` and password as the password specified in the `docker` command in step 1 above.
5. Click on "Search & Reporting" on the left sidebar.
6. In the search box, enter * (asterisk) and click the search button. There should be many new events now.
7. In the search box, enter the following query to see the mock cloud records:

```
index=main com.record_type=mock
```

   There should 10 records in the results.

8. In the search box, enter the following query to see the mock events:

```
index=main com.record_type=mock_event
```

   There should be 4 events in the results.

9. In the search box, enter the following query to see the event description fields in a table format:

```
index=main com.record_type=mock_event | table com.description
```

11.1.6 Alert Plugins

All of the store plugins discussed above can also be used as alert plugins. Additionally, there are a few plugins that are specialized as alert plugins only and do not serve very well as store plugins. Only these plugins are discussed in this section.

EmailAlert

The `EmailAlert` plugin can be used to send events to email recipients via SMTP.

This plugin is offered by the *`cloudmarker.alerts.emailalert.EmailAlert`* plugin class.

The `EmailAlert` parameters are same as that of the *`cloudmarker.util.send_email()`* function, so read its API documentation to learn about the parameters this plugin accepts.

Perform the following steps to configure Cloudmarker to send mock events as email alerts:

1. Create a config file named `cloudmarker.yaml` in the current directory with the following content:

```
plugins:
  emailalert:
    plugin: cloudmarker.alerts.emailalert.EmailAlert
    params:
      from_addr: Cloudmarker <cloudmarker@example.com>
      to_addrs:
        - user1@example.com
        - user2@example.com
      subject: Cloudmarker Alert
      host: smtp.example.com

audits:
  mockaudit:
    alerts:
      - filestore
      - emailalert
```

2. Set the values of `from_addr` and `to_addrs` appropriately.
3. If authentication is required, add `username` and `password` parameters. See `cloudmarker.send_email()` documentation for details.
4. If the SMTP host does not support `SSL`, then add `ssl_mode` parameter and set its value to `starttls` if the SMTP host supports `STARTTLS`. If the SMTP host supports neither `SSL` nor `STARTTLS`, set its value to `disable`.
5. If the SMTP host is listening on a non-standard port, then set the `port` parameter to an integer value representing the expected port number. If the SMTP host is listening on a standard port, then there is no need to set this parameter. It has a default value of `0` which automatically selects the appropriate port based on the value of `ssl_mode` parameter.
6. Run Cloudmarker with this configuration:

```
cloudmarker -n
```

7. Check the configured recipients' inboxes to confirm that the email alerts have been received.

SlackAlert

The `SlackAlert` plugin can be used to send events to Slack users via a Slack bot.

This plugin is offered by the *`cloudmarker.alerts.slackalert.SlackAlert`* plugin class.

Perform the following steps to configure Cloudmarker to send mock events as alerts via Slack:

1. Create a config file named `cloudmarker.yaml` in the current directory with the following content:

```
plugins:
  slackalert:
    plugin: cloudmarker.alerts.slackalert.SlackAlert
    params:
      bot_user_token: null
      to:
        - user1@example.com
        - user2@example.com
      text: Attention - Cloudmarker Alert

audits:
  mockaudit:
    alerts:
      - filestore
      - slackalert
```

2. Change the value of `bot_user_token` key from `null` to actual token of the Slack bot in the config file.
3. Change the vlaue of `to` key from example users to actual Slack users.
4. Now, enter this command to run Cloudmarker:

```
cloudmarker -n
```

5. The mock events would be sent to the configured Slack users as a JSON snippet.

11.1.7 Framework

Schedule

In the *built-in base config* (see *`cloudmarker.baseconfig`*), there is a `schedule` config key that specifies the local time (in 24-hour notation) at which Cloudmarker should start running audits every day. This schedule is honoured when Cloudmarker is run without the `-n` or `-now` option as follows:

```
cloudmarker
```

Logger

In the *built-in base config* (see *`cloudmarker.baseconfig`*), there is a `logger` config key that specifies an elaborate logging configuration. This can be overridden in a config file to customize the logger. For example, by default, the log files are written to `/tmp/cloudmarker.log`. If we want to override this location to, say, `log/cloudmarker.log`, we can define a config file named `cloudmarker.yaml` like this:

```
logger:
  handlers:
```

(continues on next page)

(continued from previous page)

```
    file:
      filename: log/cloudmarker.log
```

To test this configuration, enter these commands:

```
mkdir -p log
cloudmarker -n
cat log/cloudmarker.log
```

To see the default `logger` config, see *`cloudmarker.baseconfig`*. To understand more about what each of the config keys under `logger` mean, see the Python standard library logging documentation: Configuration dictionary schema.

Email

When Cloudmarker is made to run in scheduled mode, it could be useful to get email notifications about when the audits start and the audits stop. The email configuration for such audit emails can be specified under a config key named `email`. Note that this should be a top-level key in the config file, i.e., it should be at the same level as the `audits` and `run` keys.

The value for the `email` config key should be similar to the value of `params` key of an email alert. See *EmailAlert* section for more details on this. Here is an example:

```
emailalert:
  from_addr: Cloudmarker <cloudmarker@example.com>
  to_addrs:
    - user1@example.com
    - user2@example.com
  subject: Cloudmarker Alert
  host: smtp.example.com
```

With this configuration, Cloudmarker sends four types of emails:

- An email when all configured audits begin.
- An email when all configured audits end.
- An email when each configured audit begins.
- An email when each configured audit ends.

Therefore, if there are 3 audits configured under the `audits` config key, then a total of 8 emails are sent: 1 begin audits email, 1 end audits email, 3 begin audit emails (one for each audit), and 3 end audit emails (one for each audit).

CHAPTER 12

API

12.1 Cloudmarker API

12.1.1 cloudmarker package

Cloudmarker - Cloud security monitoring framework.

Subpackages

cloudmarker.alerts package

A package for alert plugins packaged with this project.

This package contains alert plugins that are packaged as part of this project. The alert plugins implement a function named `write()` that accepts input records and typically sends them to an alerting destination. The alert plugins also implement a function named `done` that perform cleanup work when called.

Note that the alert plugins implement the exact same interface as the store plugins in the *cloudmarker.stores* package. So a store plugin can usually serve equally well as an alert plugin, and vice versa. In fact, some of the store plugins such as *cloudmarker.stores.esstore.EsStore* and *cloudmarker.stores.mongodbstore.MongoDBStore* are indeed used as alert plugins too because security events can be alerted by storing them in an Elasticsearch index or MongoDB collection.

If a plugin can serve as both a store plugin and an alert plugin, we keep them in the *cloudmarker.stores* package. If a plugin makes sense only as an alert plugin, we keep them in this *cloudmarker.alerts* package.

Submodules

cloudmarker.alerts.emailalert module

Email alert plugin.

class cloudmarker.alerts.emailalert.**EmailAlert**(***kwargs*)

Bases: object

A plugin to send email alerts.

Create an instance of *EmailAlert* plugin.

This class accepts the same arguments as *cloudmarker.util.send_email()*.

The `content` argument is not honoured. Even if a `content` argument is provided, it is ignored by this class because this class defines its own content from the event records it receives in its *write()* method.

done()

Send the buffered events as an email alert.

write(*record*)

Save event record in a buffer.

Parameters **record** (*dict*) – An event record.

cloudmarker.alerts.slackalert module

Alerter to send Slack messages for identified anomalies.

class cloudmarker.alerts.slackalert.**SlackAlert**(*bot_user_token*, *to*, *text*, *temp_file='/tmp/cloudmarker/slackalert.json'*)

Bases: object

Alert plugin to send Slack alerts.

Initialize the class:*SlackAlert*.

Parameters

- **bot_user_token** (*string*) – Token for Slack bot user.
- **to** (*list*) – List of recipients (string) to send Slack alert to.
- **text** (*string*) – Message body.
- **temp_file** (*string*) – Name of file to be used to save interim JSON record which will be used to attach as report to Slack message.

done()

Write the JSON data to a file and send alert.

This function writes the JSON data to a file. The created JSON file will be used by self._post_message method to send the file as an attachment.

write(*record*)

Write records to in memory buffer.

This method will collate all the records in the list self._slack_report only.

Parameters **record** (*list*) – Records generated by Events plugin.

cloudmarker.clouds package

A package for cloud plugins packaged with this project.

This package contains cloud plugins that are packaged as part of this project. The cloud plugins implement a function named `read()` that connects to remote data sources, typically cloud APIs, and yield data records.

Submodules

cloudmarker.clouds.azcloud module

Microsoft Azure cloud plugin to read Azure infrastructure data.

This module defines the *AzCloud* class that retrieves data from Microsoft Azure.

class cloudmarker.clouds.azcloud.**AzCloud**(*tenant*, *client*, *secret*, *_max_subs=0*, *_max_recs=0*)

Bases: object

Azure cloud plugin.

Create an instance of *AzCloud* plugin.

Note: The _max_subs and _max_recs arguments should be used only in the development-test-debug phase. They should not be used in production environment. This is why we use the convention of beginning their names with underscore.

Parameters

- **tenant** (*str*) – Azure subscription tenant ID.
- **client** (*str*) – Azure service principal application ID.
- **secret** (*str*) – Azure service principal password.
- **_max_subs** (*int*) – Maximum number of subscriptions to fetch data for if the value is greater than 0.
- **_max_recs** (*int*) – Maximum number of records of each type to fetch under each subscription.

done()

Perform clean up tasks.

Currently, this method does nothing because there are no clean up tasks associated with the *AzCloud* plugin. This may change in future.

read()

Return an Azure cloud infrastructure configuration record.

Yields *dict* – An Azure cloud infrastructure configuration record.

cloudmarker.clouds.azvm module

Microsoft Azure virtual machine plugin to read Azure virtual machine data.

This module defines the *AzVM* class that retrieves virtula machine data from Microsoft Azure.

class cloudmarker.clouds.azvm.**AzVM**(*tenant*, *client*, *secret*, *_max_subs=0*, *_max_recs=0*)

Bases: object

Azure Virtual Machine plugin.

Create an instance of *AzVM* plugin.

Note: The _max_subs and _max_recs arguments should be used only in the development-test-debug phase. They should not be used in production environment. This is why we use the convention of beginning their names with underscore.

Parameters

- **tenant** (*str*) – Azure subscription tenant ID.
- **client** (*str*) – Azure service principal application ID.
- **secret** (*str*) – Azure service principal password.
- **_max_subs** (*int*) – Maximum number of subscriptions to fetch data for if the value is greater than 0.
- **_max_recs** (*int*) – Maximum number of virtual machines records to fetch for each subscription.

done()
Perform clean up tasks.

Currently, this method does nothing because there are no clean up tasks associated with the *AzVM* plugin. This may change in future.

read()
Return an Azure virtual machine record.

Yields *dict* – An Azure virtual machine record.

cloudmarker.clouds.gcpcloud module

Google Cloud Platform (GCP) plugin to read GCP infrastructure data.

This module defines the *GCPCloud* class that retrieves data from Google Cloud Platform.

class cloudmarker.clouds.gcpcloud.**GCPCloud**(*key_file_path*, *zone*)
Bases: object

GCP cloud plugin.

Create an instance of *GCPCloud* plugin.

Parameters

- **key_file_path** (*str*) – Path of the service account key file for a project.
- **zone** (*str*) – Zone of GCP Project, e.g., us-east1-b.

done()
Perform clean up tasks.

Currently, this method does nothing because there are no clean up tasks associated with the *GCPCloud* plugin. This may change in future.

read()
Return a GCP infrastructure configuration record.

Yields *dict* – Firewall rule or VM instance configuration data.

cloudmarker.clouds.mockcloud module

Mock cloud plugin for testing purpose.

class cloudmarker.clouds.mockcloud.**MockCloud**(*record_count=10*, *record_types=('foo', 'bar')*)

Bases: object

Mock cloud plugin for testing purpose.

Create an instance of *MockCloud* plugin.

This plugin generates mock records. The records generated contains three fields under three top-level keys that we also call "bucket keys": raw, data, and type, as shown in the example below:

Example

Here is an example that shows that the records generated by this plugin with the default initialization parameters:

```
>>> from cloudmarker.clouds import mockcloud
>>> cloud = mockcloud.MockCloud()
>>> for record in cloud.read():
...     print(record['raw']['data'],
...           record['ext']['record_type'],
...           record['com']['record_type'])
...
0 foo mock
1 bar mock
2 foo mock
3 bar mock
4 foo mock
5 bar mock
6 foo mock
7 bar mock
8 foo mock
9 bar mock
```

The three top-level keys, raw, ext, and com represent the names of the three buckets under which various data attributes are kept. While this is only a mock plugin, but in an actual cloud plugin implementation, the meaning of these buckets are as follows:

- raw: The value for the raw key is a dict object that represents the actual data object obtained from a cloud in its original form. No modifications should be done to the object obtained from the cloud.
- ext: The value for the ext key is a dict object which contains key-value pairs for any additional cloud-specific metadata that need to be stored. The data in this bucket is also known as *extended metadata*.
- com: The value for the com key is a dict object which contains key-value pairs for any metadata that is common to all clouds.

Parameters

- **record_count** (*int*) – Number of mock records to generate.
- **record_types** (*tuple*) – A tuple of strings that represent the different record types to be generated.

done()

Perform cleanup work.

Since this is a mock plugin, this method does nothing. However, a typical cloud plugin may or may not need to perform cleanup work in this method depending on its nature of work.

read()
Generate a record.

This method creates and yields mock records.

In actual cloud implementations, this method would typically connect to the cloud, retrieve JSON objects using the cloud API, and yield those objects as `dict` objects.

Yields *dict* – Mock record.

cloudmarker.events package

A package for event plugins packaged with this project.

This package contains event plugins that are packaged as part of this project. The event plugins implement a function named `eval` that accepts one record as parameter, evaluates the record, and generates zero or more event records for each input record. The event plugins also implement and a function named `done` that perform cleanup work when called.

Submodules

cloudmarker.events.azvmdatadiskencryptionevent module

Microsoft Azure VM Data disk encryption event.

This module defines the *AzVMDataDiskEncryptionEvent* class that identifies an unencrypted Azure VM data disk. This plugin works on the virtual machine properties found in the `com` bucket of `virtual_machine` records.

class cloudmarker.events.azvmdatadiskencryptionevent.**AzVMDataDiskEncryptionEvent**
Bases: `object`

Az VM Data disk encryption event plugin.

Create an instance of *AzVMDataDiskEncryptionEvent*.

done()
Perform cleanup work.

Currently, this method does nothing. This may change in future.

eval(*record*)
Evaluate Azure virtual machine to check for unencrypted data disks.

Parameters **record** (*dict*) – A virtual machine record.

Yields *dict* – An event record representing an unencrypted data disk of an Azure virtual machine

cloudmarker.events.azvmosdiskencryptionevent module

Microsoft Azure VM OS disk encryption event.

This module defines the *AzVMOSDiskEncryptionEvent* class that identifies an unencrypted Azure OS disk. This plugin works on the virtual machine properties found in the `com` bucket of `virtual_machine` records.

class cloudmarker.events.azvmosdiskencryptionevent.**AzVMOSDiskEncryptionEvent**
Bases: `object`

Az VM OS disk encryption event plugin.

Create an instance of *AzVMOSDiskEncryptionEvent*.

done()
Perform cleanup work.

Currently, this method does nothing. This may change in future.

eval(*record*)
Evaluate Azure virtual machine to check for unencrypted OS disk.

Parameters **record** (*dict*) – A virtual machine record.

Yields *dict* – An event record representing an unencrypted OS disk of an Azure virtual machine

cloudmarker.events.firewallruleevent module

Firewall rule event.

This module defines the *FirewallRuleEvent* class that identifies weak firewall rules. This plugin works on the firewall properties found in the `com` bucket of firewall rule records.

class cloudmarker.events.firewallruleevent.**FirewallRuleEvent**(*ports=None*)
Bases: `object`

Firewall rule event plugin.

Create an instance of *FirewallRuleEvent* plugin.

Parameters **ports** (*list*) – A list of strings that represent the ports to be checked for insecure exposure to the Internet. If `None` is specified or if unspecified, then this plugin defaults to checking ports 22, 3389, 1433, 1521, 3306, and 5432 for insecure exposure.

done()
Perform cleanup work.

Currently, this method does nothing. This may change in future.

eval(*record*)
Evaluate firewall rules to check for insecurely exposed ports.

Parameters **record** (*dict*) – A firewall rule record.

Yields *dict* – An event record representing an insecurely exposed port.

cloudmarker.events.mockevent module

Mock event plugin for testing purpose.

class cloudmarker.events.mockevent.**MockEvent**(*n=3*)
Bases: `object`

Mock event plugin for testing purpose.

Create an instance of *MockEvent* plugin.

This plugin events if the `data` field of a mock record is a multiple of `n`.

Parameters **n** (*int*) – A number that the record data value in mock record must be a multiple of in order to generate an event record.

done()
Perform cleanup work.

Since this is a mock plugin, this method does nothing. However, a typical event plugin may or may not need to perform cleanup work in this method depending on its nature of work.

eval(*record*)
Evaluate record to check for multiples of `n`.

If `record['raw']['data']` is a multiple of `n` (the parameter with which this plugin was initialized with), then generate an event record. Otherwise, do nothing.

If `record['raw']['data]` is missing, i.e., the key named `raw` or `data` does not exist, then its record number is assumed to be `1`.

This is a mock example of a event plugin. In actual event plugins, this method would typically check for security issues in the `record`.

Parameters **record** (*dict*) – Record to evaluate.

Yields *dict* – Event record if evaluation rule matches the input record.

cloudmarker.stores package

A package for store plugins packaged with this project.

This package contains store plugins that are packaged as part of this project. The store plugins implement a function named `write()` that accepts input records and typically stores them into a persistent data store. The event plugins also implement and a function named `done` that perform cleanup work when called.

Submodules

cloudmarker.stores.esstore module

Elasticsearch store plugin.

class cloudmarker.stores.esstore.**EsStore**(*host='localhost'*, *port=9200*, *index='cloudmarker'*, *buffer_size=5000000*)
Bases: `object`

Elasticsearch adapter to index cloud data in Elasticsearch.

Create an instance of *EsStore* plugin.

The plugin uses the default port for Elasticsearch if not specified.

The `buffer_size` for the plugin is the value for the maximum number of bytes of data to be sent in a bulk API request to Elasticsearch.

Parameters

- **host** (*str*) – Elasticsearch host
- **port** (*int*) – Elasticsearch port
- **index** (*str*) – Elasticsearch index
- **buffer_size** (*int*) – Maximum number of bytes of data to hold in the in-memory buffer.

done()
Flush pending records to Elasticsearch.

write(*record*)
Write JSON records to the Elasticsearch index.

Flush the buffer by saving its content to Elasticsearch when the buffer size exceeds the configured size.

Parameters **record** (*dict*) – Data to save to Elasticsearch.

cloudmarker.stores.filestore module

Filesystem store plugin.

class cloudmarker.stores.filestore.**FileStore**(*path='/tmp/cloudmarker'*)
Bases: object

A plugin to store records on the filesystem.

Create an instance of *FileStore* plugin.

Parameters **path** (*str*) – Path of directory where files are written to.

done()
Perform final cleanup tasks.

This method is called after all records have been written. In this example implementation, we properly terminate the JSON array in the .tmp file. Then we rename the .tmp file to .json file.

Note that other implementations of a store may perform tasks like closing a connection to a remote store or flushing any remaining records in a buffer.

write(*record*)
Write JSON records to the file system.

This method is called once for every `record` read from a cloud. In this example implementation of a store, we simply write the `record` in JSON format to a file. The list of records is maintained as JSON array in the file. The origin worker name in `record['com']['origin_worker']` is used to determine the filename.

The records are written to a `.tmp` file because we don't want to delete the existing complete and useful `.json` file prematurely.

Note that other implementations of a store may choose to buffer the records in memory instead of writing each record to the store immediately. They may then flush the buffer to the store based on certain conditions such as buffer size, time interval, etc.

Parameters **record** (*dict*) – Data to write to the file system.

cloudmarker.stores.mongodbstore module

MongoDB store plugin.

class cloudmarker.stores.mongodbstore.**MongoDBStore**(*host='localhost'*, *port=27017*, *db='cloudmarker'*, *collection='cloudmarker'*, *username=None*, *password=None*, *buffer_size=1000*)
Bases: object

A plugin to store records on MongoDB.

Create an instance of *MongoDBStore* plugin.

It will use the default port for mongodb 27017 if not specified. The Authentication scheme will be negotiated by MongoDB and the client for v4.0+ to SCRAM-SHA-1 or SCRAM-SHA-256 by default aftere negotiation.

Parameters

- **host** (*str*) – hostname for the DB server
- **port** (*int*) – port for mongoDB is listening
- **db** (*str*) – name of the database
- **collection** (*str*) – Name of MongoDB collection.
- **username** (*str*) – username for the database
- **password** (*str*) – password for username to authenticate with the db
- **buffer_size** (*int*) – maximum number of records to buffer

done()
Flush pending records to MongoDB and close MongoDB client.

write(*record*)
Write JSON records to the MongoDB collections.

This method is called once for every `record` read from a cloud. This method saves the records into in-memory buffers. A separate buffer is created and maintained for each record type found in `record['record_type']`. When the number of records in a buffer equals or exceeds the buffer size specified while creating an instance of *MongoDBStore* plugin, the records in the buffer are flushed (saved into a MongoDB collection).

The record type, i.e., `record['record_type']` is used to determine the collection name in MongoDB.

Parameters **record** (*dict*) – Data to save in MongoDB.

cloudmarker.stores.splunkhecstore module

SplunkStore plugin to index data in Splunk using HEC token.

class cloudmarker.stores.splunkhecstore.**SplunkHECStore**(*uri*, *token*, *index*, *ca_cert*, *buffer_size=1000*)

Bases: object

SplunkHECStore plugin to index cloud data in Splunk using HEC token.

Create an instance of *SplunkHECStore* plugin.

Parameters

- **uri** (*str*) – Splunk collector service URI.
- **token** (*str*) – Splunk HEC token.
- **index** (*str*) – Splunk HEC token accessible index.
- **ca_cert** (*str*) – Location of cetificate file to verify the identity of host in URI, or False to disable verification
- **buffer_size** (*int*) – Maximum number of records to hold in in-memory buffer for each record type.

done()
Flush any remaining records.

write(*record*)
Save the record in a bulk-buffer.

Also, flush the buffer by saving its content to Splunk when the buffer size exceeds configured self._buffer_size

Parameters **record** (*dict*) – Data to save to the Splunk.

Submodules

cloudmarker.baseconfig module

Base configuration.

cloudmarker.baseconfig.**config_yaml**
Base configuration as YAML code.

Type str

cloudmarker.baseconfig.**config_dict**
Base configuration as Python dictionary.

Type dict

Here is the complete base configuration present as a string in the *config_yaml* attribute:

```
# Base configuration
plugins:
  mockcloud:
    plugin: cloudmarker.clouds.mockcloud.MockCloud

  filestore:
    plugin: cloudmarker.stores.filestore.FileStore

  esstore:
    plugin: cloudmarker.stores.esstore.EsStore

  mongodbstore:
    plugin: cloudmarker.stores.mongodbstore.MongoDBStore

  firewallruleevent:
    plugin: cloudmarker.events.firewallruleevent.FirewallRuleEvent

  azvmosdiskencryptionevent:
    plugin: cloudmarker.events.azvmosdiskencryptionevent.AzVMOSDiskEncryptionEvent

  azvmdatadiskencryptionevent:
    plugin: cloudmarker.events.azvmdatadiskencryptionevent.AzVMDataDiskEncryptionEvent

  mockevent:
    plugin: cloudmarker.events.mockevent.MockEvent

audits:
  mockaudit:
    clouds:
      - mockcloud
    stores:
      - filestore
    events:
```

(continues on next page)

(continued from previous page)

```
      - mockevent
    alerts:
      - filestore

run:
  - mockaudit

logger:
  version: 1

  disable_existing_loggers: false

  formatters:
    simple:
      format: >-
          %(asctime)s [%(process)s] %(levelname)s
          %(name)s:%(lineno)d - %(message)s
      datefmt: "%Y-%m-%d %H:%M:%S"

  handlers:
    console:
      class: logging.StreamHandler
      formatter: simple
      stream: ext://sys.stdout

    file:
      class: logging.handlers.TimedRotatingFileHandler
      formatter: simple
      filename: /tmp/cloudmarker.log
      when: midnight
      encoding: utf8
      backupCount: 5

  loggers:
    adal-python:
      level: WARNING

  root:
    level: INFO
    handlers:
      - console
      - file

schedule: "00:00"
```

cloudmarker.manager module

Manager of worker subprocesses.

This module invokes the worker subprocesses that perform the cloud security monitoring tasks. Each worker subprocess wraps around a cloud, store, event, or alert plugin and executes the plugin in a separate subprocess.

class cloudmarker.manager.**Audit**(*audit_key*, *audit_version*, *config*)

Bases: object

Audit manager.

This class encapsulates a set of worker subprocesses and worker input queues for a single audit configuration.

Create an instance of *Audit* from configuration.

A single audit definition (from a list of audit definitions under the `audits` key in the configuration) is instantiated. Each audit definition contains lists of cloud plugins, store plugins, event plugins, and alert plugins. These plugins are instantiated and multiprocessing queues are set up to take records from one plugin and feed them to another plugin as per the audit workflow.

Parameters

- **audit_key** (*str*) – Key name for an audit configuration. This key is looked for in `config['audits']`.
- **audit_version** (*str*) – Audit version string.
- **config** (*dict*) – Configuration dictionary. This is the entire configuration dictionary that contains top-level keys named `clouds`, `stores`, `events`, `alerts`, `audits`, `run`, etc.

join()
Wait until all workers terminate.

start()
Start audit by starting all workers.

cloudmarker.manager.**main**()
Run the framework based on the schedule.

cloudmarker.util module

Utility functions.

exception cloudmarker.util.**PluginError**
Bases: `Exception`

Represents an error while loading a plugin.

exception cloudmarker.util.**PluralizeError**
Bases: `Exception`

Represents an error while converting a word to plural form.

cloudmarker.util.**expand_port_ranges**(*port_ranges*)
Expand `port_ranges` to a `set` of ports.

Examples

Here is an example usage of this function:

```
>>> from cloudmarker import util
>>> ports = util.expand_port_ranges(['22', '3389', '8080-8085'])
>>> print(ports == {22, 3389, 8080, 8081, 8082, 8083, 8084, 8085})
True
>>> ports = util.expand_port_ranges(['8080-8084', '8082-8086'])
>>> print(ports == {8080, 8081, 8082, 8083, 8084, 8085, 8086})
True
```

Note that in a port range of the form `m-n`, both `m` and `n` are included in the expanded port set. If `m > n`, we get an empty port set.

```
>>> ports = util.expand_port_ranges(['8085-8080'])
>>> print(ports == set())
True
```

If an invalid port range is found, it is ignored.

```
>>> ports = util.expand_port_ranges(['8080', '8081a', '8082'])
>>> print(ports == {8080, 8082})
True
>>> ports = util.expand_port_ranges(['7070-7075', '8080a-8085'])
>>> print(ports == {7070, 7071, 7072, 7073, 7074, 7075})
True
```

Parameters **port_ranges** (*list*) – A list of strings where each string is a port number (e.g., `'80'`) or port range (e.g., `80-89`).

Returns

A set of integers that represent the ports specified by `port_ranges`.

Return type set

`cloudmarker.util.`**`friendly_list`**(*items*, *conjunction='and'*)

Translate a list of items to a human-friendly list of items.

Examples

Here are a few example usages of this function:

```
>>> from cloudmarker import util
>>> util.friendly_list([])
'none'
>>> util.friendly_list(['apple'])
'apple'
>>> util.friendly_list(['apple', 'ball'])
'apple and ball'
>>> util.friendly_list(['apple', 'ball', 'cat'])
'apple, ball, and cat'
>>> util.friendly_list(['apple', 'ball'], 'or')
'apple or ball'
>>> util.friendly_list(['apple', 'ball', 'cat'], 'or')
'apple, ball, or cat'
```

Parameters **items** (*list*) – List of items.

Returns

Human-friendly list of items with correct placement of comma and conjunction.

Return type str

`cloudmarker.util.`**`friendly_string`**(*technical_string*)

Translate a technical string to a human-friendly phrase.

In most of our code, we use succint strings to express various technical details, e.g., `'gcp'` to express Google Cloud Platform. However these technical strings are not ideal while writing human-friendly messages such as a description of a security issue detected or a recommendation to remediate such an issue.

This function helps in converting such technical strings into human-friendly phrases that can be used in strings intended to be read by end users (e.g., security analysts responsible for protecting their cloud infrastructure) of this project.

Examples

Here are a few example usages of this function:

```
>>> from cloudmarker import util
>>> util.friendly_string('azure')
'Azure'
>>> util.friendly_string('gcp')
'Google Cloud Platform (GCP)'
```

Parameters **technical_string** (*str*) – A technical string.

Returns

Human-friendly string if a translation from a technical string to friendly string exists; the same string otherwise.

Return type str

cloudmarker.util.**load_config**(*config_paths*)

Load configuration from specified configuration paths.

Parameters **config_paths** (*list*) – Configuration paths.

Returns A dictionary of configuration key-value pairs.

Return type dict

cloudmarker.util.**load_plugin**(*plugin_config*)

Construct an object with specified plugin class and parameters.

The `plugin_config` parameter must be a dictionary with the following keys:

- `plugin`: The value for this key must be a string that represents the fully qualified class name of the plugin. The fully qualified class name is in the dotted notation, e.g., `pkg.module.ClassName`.
- `params`: The value for this key must be a `dict` that represents the parameters to be passed to the `__init__` method of the plugin class. Each key in the dictionary represents the parameter name and each value represents the value of the parameter.

Example

Here is an example usage of this function:

```
>>> from cloudmarker import util
>>> plugin_config = {
...     'plugin': 'cloudmarker.clouds.mockcloud.MockCloud',
...     'params': {
...         'record_count': 4,
...         'record_types': ('baz', 'qux')
...     }
... }
...
>>> plugin = util.load_plugin(plugin_config)
```

(continues on next page)

(continued from previous page)

```
>>> print(type(plugin))
<class 'cloudmarker.clouds.mockcloud.MockCloud'>
>>> for record in plugin.read():
...     print(record['raw']['data'],
...           record['ext']['record_type'],
...           record['com']['record_type'])
...
0 baz mock
1 qux mock
2 baz mock
3 qux mock
```

Parameters **plugin_config** (*dict*) – Plugin configuration dictionary.

Returns An object of type mentioned in the `plugin` parameter.

Return type object

Raises *PluginError* – If plugin class name is invalid.

`cloudmarker.util.`**merge_dicts**(**dicts*)

Recursively merge dictionaries.

The input dictionaries are not modified. Given any number of dicts, deep copy and merge into a new dict, precedence goes to key value pairs in latter dicts.

Example

Here is an example usage of this function:

```
>>> from cloudmarker import util
>>> a = {'a': 'apple', 'b': 'ball'}
>>> b = {'b': 'bat', 'c': 'cat'}
>>> c = util.merge_dicts(a, b)
>>> print(c == {'a': 'apple', 'b': 'bat', 'c': 'cat'})
True
```

Parameters ***dicts** (*dict*) – Variable length dictionary list

Returns Merged dictionary

Return type dict

`cloudmarker.util.`**parse_cli**(*args=None*)

Parse command line arguments.

Parameters **args** (*list*) – List of command line arguments.

Returns Parsed command line arguments.

Return type argparse.Namespace

`cloudmarker.util.`**pluralize**(*count*, *word*, **suffixes*)

Convert `word` to plural form if `count` is not `1`.

Examples

In the simplest form usage, this function just adds an `'s'` to the input word when the plural form needs to be used.

```
>>> from cloudmarker import util
>>> util.pluralize(0, 'apple')
'apples'
>>> util.pluralize(1, 'apple')
'apple'
>>> util.pluralize(2, 'apple')
'apples'
```

The plural form of some words cannot be formed merely by adding an `'s'` to the word but requires adding a different suffix. For such cases, provide an additional argument that specifies the correct suffix.

```
>>> util.pluralize(0, 'potato', 'es')
'potatoes'
>>> util.pluralize(1, 'potato', 'es')
'potato'
>>> util.pluralize(2, 'potato', 'es')
'potatoes'
```

The plural form of some words cannot be formed merely by adding a suffix but requires removing a suffix and then adding a new suffix. For such cases, provide two additional arguments: one that specifies the suffix to remove from the input word and another to specify the suffix to add.

```
>>> util.pluralize(0, 'sky', 'y', 'ies')
'skies'
>>> util.pluralize(1, 'sky', 'y', 'ies')
'sky'
>>> util.pluralize(2, 'sky', 'y', 'ies')
'skies'
```

Returns

The input `word` itself if `count` is `1`; plural form of the `word` otherwise.

Return type str

cloudmarker.util.**send_email**(*from_addr*, *to_addrs*, *subject*, *content*, *host=''*, *port=0*, *ssl_mode='ssl'*, *username=''*, *password=''*, *debug=0*)

Send email message.

When `ssl_mode` is `'ssl'` and `host` is uspecified or specified as `''` (the default), the local host is used. When `ssl_mode` is `'ssl'` and `port` is unspecified or specified as `0`, the standard SMTP-over-SSL port, i.e., port 465, is used. See `smtplib.SMTP_SSL` documentation for more details on this.

When `ssl_mode` is `'ssl'` and if `host` or `port` are unspecified, i.e., if host or port are `''` and/or `0`, respectively, the OS default behavior is used. See `smtplib.SMTP` documentation for more details on this.

We recommend these parameter values:

- Leave `ssl_mode` unspecified (thus `'ssl'` by default) if your SMTP server supports SSL.
- Set `ssl_mode` to `'starttls'` explicitly if your SMTP server does not support SSL but it supports STARTTLS.
- Set `ssl_mode` to `'disable'` explicitly if your SMTP server supports neither SSL nor STARTTLS.

- Set `host` to the SMTP hostname or address explicitly.
- Leave `port` unspecified (thus `0` by default), so that the appropriate port is chosen automatically.

With these recommendations, this function should do the right thing automatically, i.e., connect to port 465 if `use_ssl` is unspecified or `False` and port 25 if `use_ssl` is `True`.

Note that in case of SMTP, there are two different encryption protocols in use:

- SSL/TLS (or implicit SSL/TLS): SSL/TLS is used from the beginning of the connection. This occurs typically on port 465. This is enabled by default (`ssl_mode` as `'ssl'`).
- STARTTLS (or explicit SSL/TLS): The SMTP session begins as a plaintext session. Then the client (this function in this case) makes an explicit request to switch to SSL/TLS by sending the `STARTTLS` command to the server. This occurs typically on port 25 or port 587. Set `ssl_mode` to `'starttls'` to enable this behaviour

If `username` is unspecified or specified as an empty string, no SMTP authentication is done. If `username` is specified as a non-empty string, then SMTP authentication is done.

Parameters

- **from_addr** (*str*) – Sender's email address.
- **to_addrs** (*list*) – A list of `str` objects where each `str` object is a recipient's email address.
- **subject** (*str*) – Email subject.
- **content** (*str*) – Email content.
- **host** (*str*) – SMTP host.
- **port** (*int*) – SMTP port.
- **ssl_mode** (*str*) – SSL mode to use: `'ssl'` for SSL/TLS connection (the default), `'starttls'` for STARTTLS, and `'disable'` to disable SSL.
- **username** (*str*) – SMTP username.
- **password** (*str*) – SMTP password.
- **debug** (*int or bool*) – Debug level to pass to `SMTP.set_debuglevel()` to debug an SMTP session. Set to `0` (the default) or `False` to disable debugging. Set to `1` or `True` to see SMTP messages. Set to `2` to see timestamped SMTP messages.

`cloudmarker.util.`**wrap_paragraphs**(*text*, *width=70*)

Wrap each paragraph in `text` to the specified `width`.

If the `text` is indented with any common leading whitespace, then that common leading whitespace is removed from every line in text. Further, any remaining leading and trailing whitespace is removed. Finally, each paragraph is wrapped to the specified `width`.

Parameters **width** (*int*) – Maximum length of wrapped lines.

cloudmarker.workers module

Worker functions.

The functions in this module wrap around plugin classes such that these worker functions can be specified as the `target` parameter while launching a new subprocess with `multiprocessing.Process`.

Each worker function can run as a separate subprocess. While wrapping around a plugin class, each worker function creates the multiprocessing queues necessary to pass records from one plugin class to another.

cloudmarker.workers.**alert_worker**(*audit_key*, *audit_version*, *plugin_key*, *plugin*, *input_queue*)
Worker function for alert plugins.

This function behaves like *cloudmarker.workers.store_worker()*. See its documentation for details.

Parameters

- **audit_key** (*str*) – Audit key name in configuration.
- **audit_version** (*str*) – Audit version string.
- **plugin_key** (*str*) – Plugin key name in configuration.
- **plugin** (*object*) – Alert plugin object.
- **input_queue** (*multiprocessing.Queue*) – Queue to read records from.

cloudmarker.workers.**cloud_worker**(*audit_key*, *audit_version*, *plugin_key*, *plugin*, *output_queues*)
Worker function for cloud plugins.

This function expects the `plugin` object to implement a `read` method that yields records. This function calls this `read` method to retrieve records and puts each record into each queue in `output_queues`.

Parameters

- **audit_key** (*str*) – Audit key name in configuration.
- **audit_version** (*str*) – Audit version string.
- **plugin_key** (*str*) – Plugin key name in configuration.
- **plugin** (*object*) – Cloud plugin object.
- **output_queues** (*list*) – List of `multiprocessing.Queue` objects to write records to.

cloudmarker.workers.**event_worker**(*audit_key*, *audit_version*, *plugin_key*, *plugin*, *input_queue*, *output_queues*)
Worker function for event plugins.

This function expects the `plugin` object to implement a `eval` method that accepts a single record as a parameter and yields one or more records, and a `done` method to perform cleanup work in the end.

This function gets records from `input_queue` and passes each record to the `eval` method of `plugin`. Then it puts each record yielded by the `eval` method into each queue in `output_queues`.

When there are no more records in the `input_queue`, i.e., once `None` is found in the `input_queue`, this function calls the `done` method of the `plugin` to indicate that record processing is over.

Parameters

- **audit_key** (*str*) – Audit key name in configuration.
- **audit_version** (*str*) – Audit version string.
- **plugin_key** (*str*) – Plugin key name in configuration.
- **plugin** (*object*) – Store plugin object.
- **input_queue** (*multiprocessing.Queue*) – Queue to read records from.
- **output_queues** (*list*) – List of `multiprocessing.Queue` objects to write records to.

cloudmarker.workers.**store_worker**(*audit_key*, *audit_version*, *plugin_key*, *plugin*, *input_queue*)
Worker function for store plugins.

This function expects the `plugin` object to implement a `write` method that accepts a single record as a parameter and a `done` method to perform cleanup work in the end.

This function gets records from `input_queue` and passes each record to the `write` method of `plugin`.

When there are no more records in the `input_queue`, i.e., once `None` is found in the `input_queue`, this function calls the `done` method of the `plugin` to indicate that record processing is over.

Parameters

- **audit_key** (*str*) – Audit key name in configuration.
- **audit_version** (*str*) – Audit version string.
- **plugin_key** (*str*) – Plugin key name in configuration.
- **plugin** (*object*) – Store plugin object.
- **input_queue** (*multiprocessing.Queue*) – Queue to read records from.

CHAPTER 13

Indices

- genindex
- modindex
- search

Python Module Index

C

Index

F

G

J

L

M

P

R

S

W