cbapi Documentation

Release 1.5.0

Carbon Black Developer Network

Aug 05, 2019

Contents

Release v1.5.0.

cbapi provides a straightforward interface to the Carbon Black products: CB Protection, Response, and Defense. This library provides a Pythonic layer to access the raw power of the REST APIs of all CB products, making it trivial to do the easy stuff and handling all of the "sharp corners" behind the scenes for you. Take a look:

```
>>> from cbapi.response import CbResponseAPI, Process, Binary, Sensor
>>> #
>>> # Create our CbAPI object
>>> #
>>> c = CbResponseAPI()
>>> #
>>> # take the first process that ran notepad.exe, download the binary and read the␣
↪first two bytes
>>> #
>>> c.select(Process).where('process_name:notepad.exe').first().binary.file.read(2)
'MZ'
>>> #
>>> # if you want a specific ID, you can put it straight into the .select() call:
>>> #
>>> binary = c.select(Binary, "24DA05ADE2A978E199875DA0D859E7EB")
>>> #
>>> # select all sensors that have ran notepad
>>> #
>>> sensors = set()
>>> for proc in c.select(Process).where('process_name:evil.exe'):
...     sensors.add(proc.sensor)
>>> #
>>> # iterate over all sensors and isolate
>>> #
>>> for s in sensors:
...     s.network_isolation_enabled = True
...     s.save()
```

If you're more a CB Protection fellow, then you're in luck as well:

```
>>> from cbapi.protection.models import FileInstance
>>> from cbapi.protection import CbProtectionAPI
>>> #
>>> # Create our CB Protection API object
>>> #
>>> p = CbProtectionAPI()
>>> #
>>> # Select the first file instance
>>> #
>>> fi = p.select(FileInstance).first()
>>> #
>>> # print that computer's hostname. This automatically "joins" with the Computer␣
↪API object.
>>> #
>>> fi.computer.name
u'DOMAIN\\MYHOSTNAME'
>>> #
>>> # change the policy ID
>>> #
>>> fi.computer.policyId = 3
>>> fi.computer.save()
```

As of version 1.2, cbapi now provides support for CB Defense too!

```
>>> from cbapi.psc.defense import *
>>> #
>>> # Create our CB Defense API object
>>> #
>>> p = CbDefenseAPI()
>>> #
>>> # Select any devices that have the hostname WIN-IA9NQ1GN8OI and an internal IP␣
↪address of 192.168.215.150
>>> #
>>> devices = c.select(Device).where('hostNameExact:WIN-IA9NQ1GN8OI').and_(
↪"ipAddress:192.168.215.150").first()
>>> #
>>> # Change those devices' policy into the Windows_Restrictive_Workstation policy.
>>> #
>>> for dev in devices:
>>>     dev.policyName = "Restrictive_Windows_Workstation"
>>>     dev.save()
```

CHAPTER 1

Major Features

- **Enhanced Live Response API** The new cbapi now provides a robust interface to the CB Response Live Response capability. Easily create Live Response sessions, initiate commands on remote hosts, and pull down data as necessary to make your Incident Response process much more efficient and automated.
- **Consistent API for CB Response, Protection and Defense platforms** We now support CB Response, Protection and Defense users in the same API layer. Even better, the object model is the same for both; if you know one API you can easily transition to the other. cbapi hides all the differences between the three REST APIs behind a single, consistent Python-like interface.
- **Enhanced Performance** cbapi now provides a built in caching layer to reduce the query load on the Carbon Black server. This is especially useful when taking advantage of cbapi's new "joining" features. You can transparently access, for example, the binary associated with a given process in CB Response. Since many processes may be associated with the same binary, it does not make sense to repeatedly request the same binary information from the server over and over again. Therefore cbapi now caches this information to avoid unnecessary requests.
- **Reduce Complexity** cbapi now provides a friendly - dare I say "fun" - interface to the data. This greatly improves developer productivity and lowers the bar to entry.
- **Python 3 and Python 2 compatible** Use all the new features and modules available in Python 3 with cbapi. This module is compatible with Python versions 2.6.6 and above, 2.7.x, 3.4.x, and 3.5.x.
- **Better support for multiple CB servers** cbapi now introduces the concept of Credential Profiles; named collections of URL, API keys, and optional proxy configuration for connecting to any number of CB Protection, Defense, or Response servers.

CHAPTER 2

API Credentials

The new cbapi as of version 0.9.0 enforces the use of credential files.

In order to perform any queries via the API, you will need to get the API token for your CB user. See the documentation on the Developer Network website on how to acquire the API token for CB Response, CB Protection, or CB Defense.

Once you acquire your API token, place it in one of the default credentials file locations:

- `/etc/carbonblack/credentials.response` (`credentials.protection` for CB Protection, or `credentials.defense` for CB Defense)
- `~/.carbonblack/credentials.response`
- (current working directory) `.carbonblack/credentials.response`

Credentials found in a later path will overwrite earlier ones.

The credentials are stored in INI format. The name of each credential profile is enclosed in square brackets, followed by key-value pairs providing the necessary credential information:

```
[default]
url=https://localhost
token=abcdef0123456789abcdef
ssl_verify=False

[prod]
url=https://cbserver.prod.corp.com
token=aaaaaa
ssl_verify=True

[otheruser]
url=https://localhost
token=bbbbbb
ssl_verify=False
```

The possible options for each credential profile are:

- **url**: The base URL of the CB server. This should include the protocol (https) and the hostname, and nothing else.

- **token**: The API token for the user ID. More than one credential profile can be specified for a given server, with different tokens for each.
- **ssl_verify**: True or False; controls whether the SSL/TLS certificate presented by the server is validated against the local trusted CA store.
- **org_key**: The organization key. This is required to access the PSC, and can be found in the console. The format is `123ABC45`.
- **proxy**: A proxy specification that will be used when connecting to the CB server. The format is: `http://myusername:mypassword@proxy.company.com:8001/` where the hostname of the proxy is `proxy.company.com`, port 8001, and using username/password `myusername` and `mypassword` respectively.
- **ignore_system_proxy**: If you have a system-wide proxy specified, setting this to True will force cbapi to bypass the proxy and directly connect to the CB server.

Future versions of cbapi will also provide the ability to "pin" the TLS certificate so as to provide certificate verification on self-signed or internal CA signed certificates.

Environment Variable Support

The latest cbapi for python supports specifying API credentials in the following three environment variables:

CBAPI_TOKEN the envar for holding the CbR/CbP api token or the ConnectorId/APIKEY combination for CB Defense/PSC.

The *CBAPI_URL* envar holds the FQDN of the target, a CbR , CBD, or CbD/PSC server specified just as they are in the configuration file format specified above.

The optional *CBAPI_SSL_VERIFY* envar can be used to control SSL validation(True/False or 0/1), which will default to ON when not explicitly set by the user.

CHAPTER 3

Backwards & Forwards Compatibility

The previous versions (0.8.x and earlier) of cbapi and bit9Api are now deprecated and will no longer receive updates. However, existing scripts will work without change as cbapi includes both in its legacy package. The legacy package is imported by default and placed in the top level cbapi namespace when the cbapi module is imported on a Python 2.x interpreter. Therefore, scripts that expect to import cbapi.CbApi will continue to work exactly as they had previously.

Since the old API was not compatible with Python 3, the legacy package is not importable in Python 3.x and therefore legacy scripts cannot run under Python 3.

Once cbapi 1.0.0 is released, the old `cbapi.legacy.CbApi` will be deprecated and removed entirely no earlier than January 2017. New scripts should use the `cbapi.response.rest_api.CbResponseAPI` (for CB Response), `cbapi.protection.rest_api.CbProtectionAPI` (for CB Protection), or `cbapi.defense.rest_api.CbDefenseAPI` API entry points.

The API is frozen as of version 1.0; afterward, any changes in the 1.x version branch will be additions/bug fixes only. Breaking changes to the API will increment the major version number (2.x).

CHAPTER 4

User Guide

Let's get started with cbapi. Once you've mastered the concepts here, then you can always hop over to the API Documentation (below) for detailed information on the objects and methods exposed by cbapi.

4.1 Installation

Before installing cbapi, make sure that you have access to a working CB Response or CB Protection server. The server can be either on-premise or in the cloud. CB Response clusters are also supported. Once you have access to a working can use the standard Python packaging tools to install cbapi on your local machine.

Feel free to follow along with this document or watch the Development Environment Setup video on the Developer Network website.

If you already have Python installed, you can skip right down to "Using Pip".

4.1.1 Installing Python

Obviously the first thing you'll need to do is install Python on your workstation or server. We recommend using the latest version of Python 3 (as of this writing, 3.6.4) for maximum performance and compatibility. Linux and Mac OS X systems will most likely have Python installed; it will have to be installed on Windows separately.

Note that cbapi is compatible with both Python 2.7 and Python 3.x. If you already have Python 3 installed on your system, you're good to go!

If you believe you have Python installed already, run the following two commands at a command prompt:

```
$ python --version
Python 3.6.4

$ pip --version
pip 9.0.1 from /usr/local/lib/python3.6/site-packages (python 3.6)
```

If "python" reports back a version of 2.6.x, 2.7.x, or 3.x.x, you're in luck. If "pip" is not found, don't worry, we'll install that shortly.

If you're on Windows, and Python is not installed yet, download the latest Python installer from the python.org website. We recommend using the latest version of Python 3. As of this writing, the latest version available is 3.6.4. The direct link for the Python 3.6.4 installer for Windows 64-bit platforms is https://www.python.org/ftp/python/3.6.4/python-3.6.4-amd64.exe.

Ensure that the "Add Python to PATH" option is checked.

If for some reason you do not have pip installed, follow the instructions at this handy guide.

4.1.2 Using Pip

Once Python and Pip are installed, then open a command prompt and type:

```
$ pip install cbapi
```

This will download and install the latest version of cbapi from the Python PyPI packaging server.

4.1.3 Getting the Source Code

cbapi is actively developed on GitHub and the code is available from the carbonblack GitHub repository. The version of cbapi on GitHub will reflect the latest development version of cbapi and may contain bugs not present in the currently released version. On the other hand, it may contain exactly the goodies you're looking for (or you'd like to contribute back; we are happy to accept pull requests!)

To clone the latest version of the cbapi repository from GitHub:

```
$ git clone https://github.com/carbonblack/cbapi-python.git
```

Once you have a copy of the source, you can install it in "development" mode into your Python site-packages:

```
$ cd cbapi-python
$ python setup.py develop
```

This will link the version of cbapi-python you checked out into your Python site-packages directory. Any changes you make to the checked out version of cbapi will be reflected in your local Python installation. This is a good choice if you are thinking of changing or developing on cbapi itself.

4.2 Getting Started

First, let's make sure that your API authentication tokens have been imported into cbapi. Once that's done, then read on for the key concepts that will explain how to interact with Carbon Black APIs via cbapi.

Feel free to follow along with this document or watch the Development Environment Setup video on the Developer Network website.

4.2.1 API Authentication

CB Response and CB Protection use a per-user API secret token to authenticate requests via the API. The API token confers the same permissions and authorization as the user it is associated with, so protect the API token with the same care as a password.

To learn how to obtain the API token for a user, see the Developer Network website: there you will find instructions for obtaining an API token for CB Response and CB Protection.

Once you have the API token, cbapi helps keep your credentials secret by enforcing the use of a credential file. To encourage sharing of scripts across the community while at the same time protecting the security of our customers, cbapi strongly discourages embedding credentials in individual scripts. Instead, you can place credentials for several CB Response or CB Protection servers inside the API credential file and select which "profile" you would like to use at runtime.

To create the initial credential file, a simple-to-use script is provided. Just run the `cbapi-response`, `cbapi-protection`, or `cbapi-psc` script with the `configure` argument. On Mac OS X and Linux:

```
$ cbapi-response configure
```

Alternatively, if you're using Windows (change `c:\python27` if Python is installed in a different directory):

```
C:\> python c:\python27\scripts\cbapi-response configure
```

This configuration script will walk you through entering your API credentials and will save them to your current user's credential file location, which is located in the `.carbonblack` directory in your user's home directory.

If using cbapi-psc, you will also be asked to provide an org key. An org key is required to access the PSC, and can be found in the console under Settings -> API Keys.

4.2.2 Your First Query

Now that you have cbapi installed and configured, let's run a simple query to make sure everything is functional:

```
$ python
Python 2.7.10 (default, Jun 22 2015, 12:25:23)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from cbapi.response import *
>>> c = CbResponseAPI()
>>> print(c.select(Process).first().cmdline)
C:\Windows\system32\services.exe
```

That's it! Now on to the next step, learning the concepts behind cbapi.

4.3 Concepts

There are a few critical concepts that will make understanding and using the cbapi easier. These concepts are explained below, and also covered in a slide deck presented at the Carbon Black regional User Exchanges in 2016. You can see the slide deck here.

At a high level, the cbapi tries to represent data in CB Response or CB Protection as Python objects. If you've worked with SQL Object-relational Mapping (ORM) frameworks before, then this structure may seem familiar – cbapi was designed to operate much like an ORM such as SQLAlchemy or Ruby's ActiveRecord. If you haven't worked with one of these libraries, don't worry! The concepts will become clear after a little practice.

4.3.1 Model Objects

Everything in cbapi is represented in terms of "Model Objects". A Model Object in cbapi represents a single instance of a specific type of data in CB Response or Protection. For example, a process document from CB Response (as seen on an Analyze Process page in the Web UI) is represented as a `cbapi.response.models.Process` Model Object. Similarly, a file instance in CB Protection is represented as a `cbapi.protection.models.FileInstance` Model Object.

Once you have an instance of a Model Object, you can access all of the data contained within as Python properties. For example, if you have a Process Model Object named `proc` and you want to print its command line (which is stored in the `cmdline` property), you would write the code:

```
>>> print(proc.cmdline)
```

This would automatically retrieve the `cmdline` attribute of the process and print it out to your screen.

The data in CB Response and Protection may change rapidly, and so a comprehensive list of valid properties is difficult to keep up-to-date. Therefore, if you are curious what properties are available on a specific Model Object, you can print that Model Object to the screen. It will dump all of the available properties and their current values. For example:

```
>>> print(binary)
cbapi.response.models.Binary:
-> available via web UI at https://cbserver/#binary/08D1631FAF39538A133D94585644D5A8
host_count            : 1
digsig_result         : Signed
observed_filename     : [u'c:\\windows\\syswow64\\appwiz.cpl']
product_version       : 6.2.9200.16384
legal_copyright       : © Microsoft Corporation. All rights reserved.
digsig_sign_time      : 2012-07-26T08:56:00Z
orig_mod_len          : 669696
is_executable_image   : False
is_64bit              : False
digsig_publisher      : Microsoft Corporation
...
```

In this example, `host_count`, `orig_mod_len`, etc. are all properties available on this Binary Model Object. Sometimes, properties are not available on every instance of a Model Object. In this case, you can use the `.get()` method to retrieve the property, and return a default value if the property does not exist on the Model Object:

```
>>> print(binary.get("product_version", "<unknown>"))
6.2.9200.16384
```

In summary, Model Objects contain all the data associated with a specific type of API call. In this example, the `cbapi.response.models.Binary` Model Object reflects all the data available via the `/api/v1/binary` API route on a CB Response server.

4.3.2 Joining Model Objects

Many times, there are relationships between different Model Objects. To make navigating these relationships easy, cbapi provides special properties to "join" Model Objects together. For example, a `cbapi.response.models.Process` Model Object can reference the `cbapi.response.models.Sensor` or `cbapi.response.models.Binary` associated with this Process.

In this case, special "join" properties are provided for you. When you use one of these properties, cbapi will automatically retrieve the associated Model Object, if necessary.

This capability may sound like a performance killer, causing many unnecessary API calls in order to gather this data. However, cbapi has extensive Model Object caching built-in, so multiple requests for the same data will be eliminated and an API request is only made if the cache does not already contain the requested data.

For example, to print the name of the Sensor Group assigned to the Sensor that ran a specific Process:

```
>>> print(proc.sensor.group.name)
Default Group
```

Behind the scenes, this makes at most two API calls: one to obtain the Sensor associated with the Process, then another to obtain the Sensor Group that Sensor is part of. If either the Sensor or Sensor Group are already present in cbapi's internal cache, the respective API call is not made and the data is returned directly from the internal cache.

In summary, some Model Objects have special "join" properties that provide easy access to related Model Objects. A list of "join" properties is included as part of the documentation for each Model Object.

4.3.3 Queries

Now that we've covered how to get data out of a specific Model Object, we now need to learn how to obtain Model Objects in the first place! To do this, we have to create and execute a Query. cbapi Queries use the same query syntax accepted by CB Response or Protection's APIs, and add a few little helpful features along the way.

To create a query in cbapi, use the `.select()` method on the CbResponseAPI or CbProtectionAPI object. Pass the Model Object type as a parameter to the `.select()` call and optionally add filtering criteria with `.where()` clauses.

Let's start with a simple query for CB Response:

```
>>> from cbapi.response import *
>>> cb = CbResponseAPI()
>>> cb.select(Process).where("process_name:cmd.exe")
<cbapi.response.rest_api.Query object at 0x1068815d0>
```

This returns a prepared Query object with the query string `process_name:cmd.exe`. Note that at this point no API calls have been made. The cbapi Query objects are "lazy" in that they are only evaluated when you use them. If you create a Query object but never attempt to retrieve any results, no API call is ever made (I suppose that answers the age-old question; if a Query object is created, but nobody uses it, it does not make a sound, after all).

What can we do with a Query? The first thing we can do is compose new Queries. Most Query types in cbapi can be "composed"; that is, you can create a new query from more than one query string. This can be useful if you have a "base" query and want to add additional filtering criteria. For example, if we take the query above and add the additional filtering criteria `(filemod:*.exe or filemod:*.dll)`, we can write:

```
>>> base_query = cb.select(Process).where("process_name:cmd.exe")
>>> composed_query = base_query.where("(filemod:*.exe or filemod:*.dll")
```

Now the `composed_query` is equivalent to a query of `process_name:cmd.exe (filemod:*.exe or filemod:*.dll)`. You can also add sorting criteria to a query:

```
>>> sorted_query = composed_query.sort("last_update asc")
```

Now when we execute the `sorted_query`, the results will be sorted by the last server update time in ascending order.

Ok, now we're ready to actually execute a query and retrieve the results. You can think of a Query as a kind of "infinite" Python list. Generally speaking, you can use all the familiar ways to access a Python list to access the results of a cbapi query. For example:

```
>>> len(base_query)     # How many results were returned for the query?
3

>>> base_query[:2]      # I want the first two results
[<cbapi.response.models.Process: id 00000003-0000-036c-01d2-2efd3af51186-00000001> @␣
↪https://cbserver,
<cbapi.response.models.Process: id 00000003-0000-07d4-01d2-2efcd4949dfc-00000001> @␣
↪https://cbserver]

>>> base_query[-1:]     # I want the last result
[<cbapi.response.models.Process: id 00000002-0000-0f2c-01d2-2a57625ca0dd-00000001> @␣
↪https://cbserver]

>>> for proc in base_query:  # Loop over all the results
>>>     print(proc.cmdline)
"C:\Windows\system32\cmd.exe"
"C:\Windows\system32\cmd.exe"
"C:\Windows\system32\cmd.exe"

>>> procs = list(base_query) # Just make a list of all the results
```

In addition to using a Query object as an array, two helper methods are provided as common shortcuts. The first method is `.one()`. The `.one()` method is useful when you know only one result should match your query; it will throw a `MoreThanOneResultError` exception if there are zero or more than one results for the query. The second method is `.first()`, which will return the first result from the result set, or None if there are no results.

Every time you access a Query object, it will perform a REST API query to the Carbon Black server. For large result sets, the results are retrieved in batches- by default, 100 results per API request on CB Response and 1,000 results per API request on CB Protection. The search queries themselves are not cached, but the resulting Model Objects are.

4.3.4 Retrieving Objects by ID

Every Model Object (and in fact any object addressable via the REST API) has a unique ID associated with it. If you already have a unique ID for a given Model Object, for example, a Process GUID for CB Response, or a Computer ID for CB Protection, you can ask cbapi to give you the associated Model Object for that ID by passing that ID to the `.select()` call. For example:

```
>>> binary = cb.select(Binary, "CA4FAFFA957C71C006B59E29DFE3EB8B")
>>> print(binary.file_desc)
PNRP Name Space Provider
```

Note that retrieving an object via `.select()` with the ID does not automatically request the object from the server via the API. If the Model Object is already in the local cache, the locally cached version is returned. If it is not, a "blank" Model Object is created and is initialized only when an attempt is made to read a property. Therefore, assuming an empty cache, in the example above, the REST API query would not happen until the second line (the `print` statement). If you want to ensure that an object exists at the time you call `.select()`, add the `force_init=True`

keyword parameter to the `.select()` call. This will cause cbapi to force a refresh of the object and if it does not exist, cbapi will throw a `ObjectNotFoundError` exception.

4.3.5 Creating New Objects

The CB Response and Protection REST APIs provide the ability to insert new data under certain circumstances. For example, the CB Response REST API allows you to insert a new banned hash into its database. Model Objects that represent these data types can be "created" in cbapi by using the `create()` method:

```
>>> bh = cb.create(BannedHash)
```

If you attempt to create a Model Object that cannot be created, you will receive a `ApiError` exception.

Once a Model Object is created, it's blank (it has no data). You will need to set the required properties and then call the `.save()` method:

```
>>> bh = cb.create(BannedHash)
>>> bh.text = "Banned from API"
>>> bh.md5sum = "CA4FAFFA957C71C006B59E29DFE3EB8B"
>>> bh.save()
```

If you don't fill out all the properties required by the API, then you will receive an `InvalidObjectError` exception with a list of the properties that are required and not currently set.

Once the `.save()` method is called, the appropriate REST API call is made to create the object. The Model Object is then updated to the current state returned by the API, which may include additional data properties initialized by CB Response or Protection.

4.3.6 Modifying Existing Objects

The same `.save()` method can be used to modify existing Model Objects if the REST API provides that capability. If you attempt to modify a Model Object that cannot be changed, you will receive a `ApiError` exception.

For example, if you want to change the "jgarman" user's password to "cbisawesome":

```
>>> user = cb.select(User, "jgarman")
>>> user.password = "cbisawesome"
>>> user.save()
```

4.3.7 Deleting Objects

Simply call the `.delete()` method on a Model Object to delete it (again, if you attempt to delete a Model Object that cannot be deleted, you will receive a `ApiError` exception).

Example:

```
>>> user = cb.select(User, "jgarman")
>>> user.delete()
```

4.3.8 Tracking Changes to Objects

Internally, Model Objects track all changes between when they were last refreshed from the server up until `.save()` is called. If you're interested in what properties have been changed or added, simply `print` the Model Object.

You will see a display like the following:

```
>>> user = cb.create(User)
>>> user.username = "jgarman"
>>> user.password = "cbisawesome"
>>> user.first_name = "Jason"
>>> user.last_name = "Garman"
>>> user.teams = []
>>> user.global_admin = False
>>> print(user)
User object, bound to https://cbserver.
 Partially initialized. Use .refresh() to load all attributes
--------------------------------------------------------------------------------

(+)                email: jgarman@carbonblack.com
(+)           first_name: Jason
(+)         global_admin: False
                    id: None
(+)            last_name: Garman
(+)             password: cbisawesome
(+)                teams: []
(+)             username: jgarman
```

Here, the `(+)` symbol before a property name means that the property will be added the next time that `.save()` is called. Let's call `.save()` and modify one of the Model Object's properties:

```
>>> user.save()
>>> user.first_name = "J"
>>> print(user)
print(user)
User object, bound to https://cbserver.
 Last refreshed at Mon Nov  7 16:54:00 2016
--------------------------------------------------------------------------------

             auth_token: 8b2dcf9d59b7da1a0b2b4ec50a77d8ca3d7dcb9c
                  email: jgarman@carbonblack.com
(*)           first_name: J
            global_admin: False
                      id: jgarman
               last_name: Garman
                   teams: []
                username: jgarman
```

The `(*)` symbol means that a property value will be changed the next time that `.save()` is called. This time, let's forget about our changes by calling `.reset()` instead:

```
>>> user.reset()
>>> print(user.first_name)
Jason
```

Now the user Model Object has been restored to the original state as it was retrieved from the server.

4.4 Logging & Diagnostics

The cbapi provides extensive logging facilities to track down issues communicating with the REST API and understand potential performance bottlenecks.

4.4.1 Enabling Logging

The cbapi uses Python's standard `logging` module for logging. To enable debug logging for the cbapi, you can do the following:

```
>>> import logging
>>> root = logging.getLogger()
>>> root.addHandler(logging.StreamHandler())
>>> logging.getLogger("cbapi").setLevel(logging.DEBUG)
```

All REST API calls, including the API endpoint, any data sent via POST or PUT, and the time it took for the call to complete:

```
>>> user.save()
Creating a new User object
Sending HTTP POST /api/user with {"email": "jgarman@carbonblack.com", "first_name":
↪"Jason", "global_admin": false, "id": null, "last_name": "Garman", "password":
↪"cbisawesome", "teams": [], "username": "jgarman"}
HTTP POST /api/user took 0.079s (response 200)
Received response: {u'result': u'success'}
HTTP GET /api/user/jgarman took 0.011s (response 200)
```

4.5 CB Response API Examples

Now that we've covered the basics, let's step through a few examples using the CB Response API. In these examples, we will assume the following boilerplate code to enable logging and establish a connection to the "default" CB Response server in our credential file:

```
>>> import logging
>>> root = logging.getLogger()
>>> root.addHandler(logging.StreamHandler())
>>> logging.getLogger("cbapi").setLevel(logging.DEBUG)

>>> from cbapi.response import *
>>> cb = CbResponseAPI()
```

With that boilerplate out of the way, let's take a look at a few examples.

4.5.1 Download a Binary from CB Response

Let's grab a binary that CB Response has collected from one of the endpoints. This can be useful if you want to send this binary for further automated analysis or pull it down for manual reverse engineering. You can see a full example with command line options in the examples directory: `binary_download.py`.

Let's step through the example:

```
>>> import shutil
>>> md5 = "7FB55F5A62E78AF9B58D08AAEEAEF848"
>>> binary = cb.select(Binary, md5)
>>> shutil.copyfileobj(binary.file, open(binary.original_filename, "wb"))
```

First, we select the binary by its primary key: the MD5 hash of the binary contents. The third line requests the binary file data by accessing the `file` property on the Binary Model Object. The `file` property acts as a read-only, Python file-like object. In this case, we use the Python `shutil` library to copy one file object to another. The advantage of

using `shutil` is that the file is copied in chunks, and the full file does not have to be read into memory before saving it to disk.

Another way to use the `file` property is to call `.read()` on it just like any other Python file object. The following code will read the first two bytes from the Binary:

```
>> binary.file.read(2)
"MZ"
```

4.5.2 Ban a Binary

Now let's take this binary and add a Banning rule for it. To do this, we create a new BannedHash Model Object:

```
>>> bh = cb.create(BannedHash)
>>> bh.md5hash = binary.md5
>>> bh.text = "Banned from API"
>>> bh.enabled = True
>>> bh.save()
Creating a new BannedHash object
Sending HTTP POST /api/v1/banning/blacklist with {"md5hash":
↪"7FB55F5A62E78AF9B58D08AAEEAEF848", "text": "banned from API"}
HTTP POST /api/v1/banning/blacklist took 0.035s (response 200)
Received response: {u'result': u'success'}
HTTP GET /api/v1/banning/blacklist/7FB55F5A62E78AF9B58D08AAEEAEF848 took 0.039s␣
↪(response 200)
```

Note that if the hash is already banned in CB Response, then you will receive a *ServerError* exception with the message that the banned hash already exists.

4.5.3 Isolate a Sensor

Switching gears, let's take a Sensor and quarantine it from the network. The CB Response network isolation functionality allows administrators to isolate endpoints that may be actively involved in an incident, while preserving access to perform Live Response on that endpoint and collect further endpoint telemetry.

To isolate a sensor, we first need to acquire its Sensor Model Object:

```
>>> sensor = cb.select(Sensor).where("hostname:HOSTNAME").first()
```

This will select the first sensor that matches the hostname `HOSTNAME`. Now we can isolate that machine:

```
>>> sensor.isolate()
Updating Sensor with unique ID 4
Sending HTTP PUT /api/v1/sensor/4 with {"boot_id": "0", "build_id": 5, "build_version_
↪string": "005.002.000.61003", ...}
HTTP PUT /api/v1/sensor/4 took 0.129s (response 204)
HTTP GET /api/v1/sensor/4 took 0.050s (response 200)
...
True
```

The `.isolate()` method will keep polling the CB Response server until the sensor has confirmed that it is now isolated from the network. If the sensor is offline or otherwise unreachable, this call could never return. Therefore, there is also a `timeout=` keyword parameter that can be used to set an optional timeout that, if reached, will throw a `TimeoutError` exception. The `.isolate()` function returns True when the sensor is successfully isolated.

When you're ready to restore full network connectivity to the sensor, simply call the `.unisolate()` method:

```
>>> sensor.unisolate()
Updating Sensor with unique ID 4
Sending HTTP PUT /api/v1/sensor/4 with {"boot_id": "0", "build_id": 5, "build_version_
↪string": "005.002.000.61003", ...}
HTTP PUT /api/v1/sensor/4 took 0.077s (response 204)
HTTP GET /api/v1/sensor/4 took 0.020s (response 200)
...
True
```

Again, once the sensor is back on the network, the `.unisolate()` method will return True. Just like `.isolate()`, you can optionally specify a timeout using the `timeout=` keyword parameter.

4.5.4 Querying Processes and Events

Now, let's do some queries into the CB Response database. The true power of CB Response is its continuous recording and powerful query language that allows you to go back in time and track the root cause of any security incident on your endpoints. Let's start with a simple query to find instances of a specific behavioral IOC, where our attacker used the built-in Windows tool `net.exe` to mount an internal network share. We will iterate over all uses of `net.exe` to mount our target share, printing out the parent processes that led to the execution of the offending command:

```
>>> query = cb.select(Process).where("process_name:net.exe").and_(r
↪"cmdline:\\test\blah").group_by("id")
>>> def print_details(proc, depth):
...     print("%s%s: %s ran %s" % (" "*depth, proc.start, proc.username, proc.
↪cmdline))
...
>>> for proc in query:
...     print_details(proc, 0)
...     proc.walk_parents(print_details)
...
HTTP GET /api/v1/process?cb.urlver=1&facet=false&q=process_name%3Anet.exe+cmdline%3A
↪%5C%5Ctest%5Cblah&rows=100&sort=last_update+desc&start=0 took 0.462s (response 200)
2016-11-11 20:59:31.631000: WIN-IA9NQ1GN8OI\bit9rad ran net  use y: \\test\blah
HTTP GET /api/v3/process/00000003-0000-036c-01d2-2efd3af51186/1/event took 0.036s␣
↪(response 200)
2016-10-25 20:20:29.790000: WIN-IA9NQ1GN8OI\bit9rad ran "C:\Windows\system32\cmd.exe"
HTTP GET /api/v3/process/00000003-0000-0c34-01d2-2ec94f09cae6/1/event took 0.213s␣
↪(response 200)
 2016-10-25 14:08:49.651000: WIN-IA9NQ1GN8OI\bit9rad ran C:\Windows\Explorer.EXE
HTTP GET /api/v3/process/00000003-0000-0618-01d2-2ec94edef208/1/event took 0.013s␣
↪(response 200)
  2016-10-25 14:08:49.370000: WIN-IA9NQ1GN8OI\bit9rad ran␣
↪C:\Windows\system32\userinit.exe
HTTP GET /api/v3/process/00000003-0000-02ec-01d2-2ec9412b4b70/1/event took 0.017s␣
↪(response 200)
   2016-10-25 14:08:26.382000: SYSTEM ran winlogon.exe
HTTP GET /api/v3/process/00000003-0000-02b0-01d2-2ec94115df7a/1/event took 0.012s␣
↪(response 200)
    2016-10-25 14:08:26.242000: SYSTEM ran \SystemRoot\System32\smss.exe 00000001␣
↪00000030
HTTP GET /api/v3/process/00000003-0000-0218-01d2-2ec93f813429/1/event took 0.021s␣
↪(response 200)
     2016-10-25 14:08:23.590000: SYSTEM ran \SystemRoot\System32\smss.exe
HTTP GET /api/v3/process/00000003-0000-0004-01d2-2ec93f7c7181/1/event took 0.081s␣
↪(response 200)
      2016-10-25 14:08:23.559000: SYSTEM ran c:\windows\system32\ntoskrnl.exe
```

(continues on next page)

(continued from previous page)

```
HTTP GET /api/v3/process/00000003-0000-0000-01d2-2ec93f6051ee/1/event took 0.011s␣
↪(response 200)
       2016-10-25 14:08:23.374000:  ran c:\windows\system32\ntoskrnl.exe
HTTP GET /api/v3/process/00000003-0000-0004-01d2-2ec93f6051ee/1/event took 0.011s␣
↪(response 200)
2016-11-11 20:59:25.667000: WIN-IA9NQ1GN8OI\bit9rad ran net  use z: \\test\blah
2016-10-25 20:20:29.790000: WIN-IA9NQ1GN8OI\bit9rad ran "C:\Windows\system32\cmd.exe"
 2016-10-25 14:08:49.651000: WIN-IA9NQ1GN8OI\bit9rad ran C:\Windows\Explorer.EXE
  2016-10-25 14:08:49.370000: WIN-IA9NQ1GN8OI\bit9rad ran␣
↪C:\Windows\system32\userinit.exe
   2016-10-25 14:08:26.382000: SYSTEM ran winlogon.exe
    2016-10-25 14:08:26.242000: SYSTEM ran \SystemRoot\System32\smss.exe 00000001␣
↪00000030
     2016-10-25 14:08:23.590000: SYSTEM ran \SystemRoot\System32\smss.exe
      2016-10-25 14:08:23.559000: SYSTEM ran c:\windows\system32\ntoskrnl.exe
       2016-10-25 14:08:23.374000:  ran c:\windows\system32\ntoskrnl.exe
```

That was a lot in one code sample, so let's break it down part-by-part.

First, we set up the `query` variable by creating a new `Query` object using the `.where()` and `.and_()` methods. Next, we define a function that will get called on each parent process all the way up the chain to the system kernel loading during the boot process. This function, `print_details`, will print a few data points about each process: namely, the local endpoint time when that process started, the user who spawned the process, and the command line for the process.

Finally, we execute our query by looping over the result set with a Python for loop. For each process that matches the query, first we print details of the process itself (the process that called `net.exe` with a command line argument of our target share `\\test\blah`), then calls the `.walk_parents()` helper method to walk up the chain of all parent processes. Each level of parent process (the "depth") is represented by an extra space; therefore, reading backwards, you can see that `ntoskrnl.exe` spawned `smss.exe`, which in turn spawned `winlogon.exe`, and so on. You can see the full backwards chain of events that ultimately led to the execution of each of these `net.exe` calls.

Remember that we have logging turned on for these examples, so you see each of the HTTP GET requests to retrieve process event details as they happen. Astute observers will note that walking the parents of the second `net.exe` command, where the `\\test\blah` share was mounted on the `z:` drive, did not trigger additional HTTP GET requests. This is thanks to cbapi's caching layer. Since both `net.exe` commands ran as part of the same command shell session, the parent processes are shared between the two executions. Since the parent processes were already requested as part of the previous walk up the chain of parent processes, cbapi did not re-request the data from the server, instead using its internal cache to satisfy the process information requests from this script.

New Filters: Group By, Time Restrictions

In the query above, there is an extra `.group_by()` method. This method is new in cbapi 1.1.0 and is part of five new query filters available when communicating with a CB Response 6.1 server. These filters are accessible via methods on the `Process` Query object. These new methods are:

- `.group_by()` - Group the result set by a field in the response. Typically you will want to group by `id`, which will ensure that the result set only has one result per *process* rather than one result per *event segment*. For more information on processes, process segments, and how segments are stored in CB Response 6.0, see the Process API Changes for CB Response 6.0 page on the Developer Network website.
- `.min_last_update()` - Only return processes that have events after a given date/time stamp (relative to the individual sensor's clock)
- `.max_last_update()` - Only return processes that have events before a given date/time stamp (relative to the individual sensor's clock)

- `.min_last_server_update()` - Only return processes that have events after a given date/time stamp (relative to the CB Response server's clock)
- `.max_last_server_update()` - Only return processes that have events before a given date/time stamp (relative to the CB Response server's clock)

CB Response 6.1 uses a new way of recording process events that greatly increases the speed and scale of collection, allowing you to store and search data for more endpoints on the same hardware. Details on the new database format can be found on the Developer Network website at the Process API Changes for CB Response 6.0 page.

The `Process` Model Object traditionally referred to a single "segment" of events in the CB Response database. In CB Response versions prior to 6.0, a single segment will include up to 10,000 individual endpoint events, enough to handle over 95% of the typical event activity for a given process. Therefore, even though a `Process` Model Object technically refers to a single *segment* in a process, since most processes had less than 10,000 events and therefore were only comprised of a single segment, this distinction wasn't necessary.

However, now that processes are split across many segments, a better way of handling this is necessary. Therefore, CB Response 6.0 introduces the new `.group_by()` method.

More on Filters

Querying for a process will return *all* segments that match. For example, if you search for `process_name:cmd.exe`, the result set will include *all* segments of *all* `cmd.exe` processes. Therefore, CB Response 6.1 introduced the ability to "group" result sets by a field in the result. Typically you will want to group by the internal process id (the `id` field), and this is what we did in the query above. Grouping by the `id` field will ensure that only one result is returned per *process* rather than per *segment*.

Let's take a look at an example:

```
>>> from datetime import datetime, timedelta
>>> yesterday = datetime.utcnow() - timedelta(days=1)       # Get "yesterday" in GMT
>>> for proc in c.select(Process).where("process_name:cmd.exe").min_last_
↪update(yesterday):
...     print proc.id, proc.segment
DEBUG:cbapi.connection:HTTP GET /api/v1/process?cb.min_last_update=2017-05-21T18%3A41
↪%3A58Z&cb.urlver=1&facet=false&q=process_name%3Acmd.exe&rows=100&sort=last_
↪update+desc&start=0 took 2.164s (response 200)
00000001-0000-0e48-01d2-c2a397f4cfe0 1495465643405
00000001-0000-0e48-01d2-c2a397f4cfe0 1495465407157
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463680155
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463807694
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463543944
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463176570
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463243492
```

Notice that the "same" process ID is returned seven times, but with seven different segment IDs. CB Response will return *every* process event segment that matches a given query, in this case, any event segment that contains the process command name `cmd.exe`.

That is, however, most likely not what you wanted. Instead, you'd like a list of the *unique* processes associated with the command name `cmd.exe`. Just add the `.group_by("id")` filter to your query:

```
>>> for proc in c.select(Process).where("process_name:cmd.exe").min_last_
↪update(yesterday).group_by("id"):
...     print proc.id, proc.segment
DEBUG:cbapi.connection:HTTP GET /api/v1/process?cb.group=id&cb.min_last_update=2017-
↪05-21T18%3A41%3A58Z&cb.urlver=1&facet=false&q=process_name%3Acmd.exe&rows=100&
↪sort=last_update+desc&start=0 took 2.163s (response 200)
```

(continues on next page)

(continued from previous page)

```
00000001-0000-0e48-01d2-c2a397f4cfe0 1495465643405
```

4.5.5 Feed and Watchlist Maintenance

The cbapi provides several helper functions to assist in creating watchlists and

Watchlists are simply saved Queries that are automatically run on the CB Response server on a periodic basis. Results of the watchlist are tagged in the database and optionally trigger alerts. Therefore, a cbapi Query can easily be converted into a watchlist through the Query `.create_watchlist()` function:

```
>>> new_watchlist = query.create_watchlist("[WARN] Attempts to mount internal share")
Creating a new Watchlist object
Sending HTTP POST /api/v1/watchlist with {"id": null, "index_type": "events", "name":
↪"[WARN] Attempts to mount internal share", "search_query": "facet=false&q=process_
↪name%3Anet.exe+cmdline%3A%5C%5Ctest%5Cblah&cb.urlver=1&sort=last_update+desc"}
HTTP POST /api/v1/watchlist took 0.510s (response 200)
Received response: {u'id': 222}
Only received an ID back from the server, forcing a refresh
HTTP GET /api/v1/watchlist/222 took 0.034s (response 200)
```

This helper function will automatically create a watchlist from the Query object with the given name.

If you have a watchlist that already exists, the Watchlist Model Object can help you extract the human-readable query from the watchlist. Just select the watchlist and access the `.query` property on the Watchlist Model Object:

```
>>> my_watchlist = cb.select(Watchlist).where("name:[WARN] Attempts to mount internal␣
↪share").one()
>>> print(my_watchlist.query)
process_name:net.exe cmdline:\\test\blah
```

You can also execute the query straight from the Watchlist Model Object:

```
>>> len(my_watchlist.search())
HTTP GET /api/v1/process?cb.urlver=1&facet=false&q=process_name%3Anet.exe+cmdline%3A
↪%5C%5Ctest%5Cblah&rows=0&start=0 took 0.477s (response 200)
2
```

And finally, you can of course enable and disable Watchlists:

```
>>> my_watchlist.enabled = False
>>> my_watchlist.save()
Updating Watchlist with unique ID 222
Sending HTTP PUT /api/v1/watchlist/222 with {"alliance_id": null, "date_added": "2016-
↪11-15 23:48:27.615993-05:00", "enabled": false, "from_alliance": false, "group_id":␣
↪-1, "id": "222", "index_type": "events", "last_hit": "2016-11-15 23:50:08.448685-
↪05:00", "last_hit_count": 2, "name": "[WARN] Attempts to mount internal share",
↪"readonly": false, "search_query": "facet=false&q=process_name%3Anet.exe%20cmdline
↪%3A%5C%5Ctest%5Cblah&cb.urlver=1", "search_timestamp": "2016-11-16T04:50:01.750240Z
↪", "total_hits": "2", "total_tags": "2"}
HTTP PUT /api/v1/watchlist/222 took 0.036s (response 200)
Received response: {u'result': u'success'}
HTTP GET /api/v1/watchlist/222 took 0.029s (response 200)
```

You can see more examples of Feed and Watchlist maintenance in the `feed_operations.py` and `watchlist_operations.py` example scripts.

4.5.6 Managing Threat Reports & Alerts

The cbapi provides helper functions to manage alerts and threat reports in bulk. The Query objects associated with the ThreatReport and Alert Model Objects provide a few bulk operations to help manage large numbers of Threat Reports and Alerts, respectively.

To mark a large number of Threat Reports as false positives, create a query that matches the Reports you're interested in. For example, if every Report from the Feed named "SOC" that contains the word "FUZZYWOMBAT" in the report title should be considered a false positive (and no longer trigger Alerts), you can write the following code to do so:

```
>>> feed = c.select(Feed).where("name:SOC").one()
>>> report_query = feed.reports.where("title:FUZZYWOMBAT")
>>> report_query.set_ignored()
```

Similar actions can be taken on Alerts. The AlertQuery object exposes three helper methods to perform bulk operations on sets of Alerts: `.set_ignored()`, `.assign_to()`, and `.change_status()`.

4.5.7 Joining Everything Together

Now that we've examined how to request information on binaries, sensors, and processes through cbapi, let's chain this all together using the "join" functionality of cbapi's Model Objects. Let's just tweak the `print_details` function from above to add a few more contextual details. Our new function will now include the following data points for each process:

- The hostname the process was executed on
- The sensor group that host belongs to
- **If the binary was signed, also print out:**
  - The number of days between when the binary was signed and it was executed on the endpoint
  - The verified publisher name from the digital signature

We can transparently "join" between the Process Model Object and the Sensor, Sensor Group, and Binary Model Objects using the appropriately named helper properties. Here's the new function:

```
>>> import pytz

>>> def print_details(proc, depth):
...     print("On host {0} (part of sensor group {1}):".format(proc.hostname, proc.
↪sensor.group.name))
...     print("- At {0}, process {1} was executed by {2}".format(proc.start, proc.
↪cmdline, proc.username))
...     if proc.binary.signed:
...         # force local timestamp into UTC, we're just looking for an estimate here.
...         utc_timestamp = proc.start.replace(tzinfo=pytz.timezone("UTC"))
...         days_since_signed = (utc_timestamp - proc.binary.signing_data.sign_time).
↪days
...         print("- That binary ({0}) was signed by {1} {2} days before it was␣
↪executed.".format(proc.process_md5,
...             proc.binary.signing_data.publisher, days_since_signed))
```

Now if we run our for loop from above again:

```
>>> for proc in query:
...     print_details(proc, 0)
```

(continues on next page)

(continued from previous page)

```
...     proc.walk_parents(print_details)
...
HTTP GET /api/v1/process?cb.urlver=1&facet=false&q=process_name%3Anet.exe+cmdline%3A
↪%5C%5Ctest%5Cblah&rows=100&sort=last_update+desc&start=0 took 0.487s (response 200)
HTTP GET /api/v1/sensor/3 took 0.037s (response 200)
HTTP GET /api/group/1 took 0.022s (response 200)
On host WIN-IA9NQ1GN8OI (part of sensor group Default Group):
- At 2016-11-11 20:59:31.631000, process net  use y: \\test\blah was executed by WIN-
↪IA9NQ1GN8OI\bit9rad
HTTP GET /api/v1/binary/79B6D4C5283FC806387C55B8D7C8B762/summary took 0.016s␣
↪(response 200)
- That binary (79b6d4c5283fc806387c55b8d7c8b762) was signed by Microsoft Corporation␣
↪1569 days before it was executed.
HTTP GET /api/v3/process/00000003-0000-036c-01d2-2efd3af51186/1/event took 0.045s␣
↪(response 200)
On host WIN-IA9NQ1GN8OI (part of sensor group Default Group):
- At 2016-10-25 20:20:29.790000, process "C:\Windows\system32\cmd.exe"  was executed␣
↪by WIN-IA9NQ1GN8OI\bit9rad
HTTP GET /api/v1/binary/BF93A2F9901E9B3DFCA8A7982F4A9868/summary took 0.015s␣
↪(response 200)
- That binary (bf93a2f9901e9b3dfca8a7982f4a9868) was signed by Microsoft Corporation␣
↪1552 days before it was executed.
```

Those few lines of Python above are jam-packed with functionality. Now for each process execution, we have added contextual information on the source host, the group that host is part of, and details about the signing status of the binary that was executed. The magic is performed behind the scenes when we use the `.binary` and `.sensor` properties on the Process Model Object. Just like our previous example, cbapi's caching layer ensures that we do not overload the CB Response server with duplicate requests for the same data. In this example, multiple redundant requests for sensor, sensor group, and binary data are all eliminated by cbapi's cache.

4.5.8 Facets

The cbapi also provides functionality to pull facet information from the database. You can use the `.facet()` method on a Query object to retrieve facet (ie. "group") information for a given query result set. Here's an example that pulls the most common process names for our sample host:

```
>>> def print_facet_histogram(facets):
...     for entry in facets:
...         print("%15s: %5s%% %s" % (entry["name"][:15], entry["ratio"], u"\u25A0
↪"*(int(entry["percent"])/2)))
...

>>> facet_query = cb.select(Process).where("hostname:WIN-IA9NQ1GN8OI").and_(
↪"username:bit9rad")
>>> print_facet_histogram(facet_query.facets("process_name")["process_name"])

HTTP GET /api/v1/process?cb.urlver=1&facet=true&facet.field=process_name&facet.
↪field=username&q=hostname%3AWIN-IA9NQ1GN8OI+username%3Abit9rad&rows=0&start=0 took␣
↪0.024s (response 200)
     chrome.exe:  23.4% ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
thumbnailextrac:  15.4% ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
   adobearm.exe:   8.6% ■■■■■■■■■■■■■■■■■
   taskhost.exe:   6.0% ■■■■■■■■■■■■
    conhost.exe:   4.7% ■■■■■■■■■
```

(continues on next page)

(continued from previous page)

```
      ping.exe:    4.0% ========
    wermgr.exe:    3.5% =======
```

In the above example, we just pulled one facet: the `process_name`; you can ask the server for faceting on multiple fields in one query by simply listing the fields in the call to `.facet()`: for example, `.facet("username", "process_name")` will produce a dictionary with two top-level keys: `username` and `process_name`.

4.5.9 Administrative Tasks

In addition to querying data, you can also perform various administrative tasks using cbapi.

Let's create a user on our CB Response server:

```
>>> user = cb.create(User)
>>> user.username = "jgarman"
>>> user.password = "cbisawesome"
>>> user.first_name = "Jason"
>>> user.last_name = "Garman"
>>> user.email = "jgarman@carbonblack.com"
>>> user.teams = []
>>> user.global_admin = False
Creating a new User object
Sending HTTP POST /api/user with {"email": "jgarman@carbonblack.com", "first_name":
↪"Jason", "global_admin": false, "id": null, "last_name": "Garman", "password":
↪"cbisawesome", "teams": [], "username": null}
HTTP POST /api/user took 0.608s (response 200)
Received response: {u'result': u'success'}
```

How about moving a sensor to a new Sensor Group:

```
>>> sg = cb.create(SensorGroup)
>>> sg.name = "Critical Endpoints"
>>> sg.site = 1
>>> sg.save()
Creating a new SensorGroup object
Sending HTTP POST /api/group with {"id": null, "name": "Critical Endpoints", "site_id
↪": 1}
HTTP POST /api/group took 0.282s (response 200)
Received response: {u'id': 2}
Only received an ID back from the server, forcing a refresh
HTTP GET /api/group/2 took 0.011s (response 200)
>>> sensor = cb.select(Sensor).where("hostname:WIN-IA9NQ1GN8OI").first()
>>> sensor.group = sg
>>> sensor.save()
Updating Sensor with unique ID 3
Sending HTTP PUT /api/v1/sensor/3 with {"boot_id": "2", "build_id": 2, "build_version_
↪string": "005.002.000.60922", ...
HTTP PUT /api/v1/sensor/3 took 0.087s (response 204)
HTTP GET /api/v1/sensor/3 took 0.030s (response 200)
```

4.6 CbAPI and Live Response

Working with the CB Live Response REST API directly can be difficult. Thankfully, just like the rest of Carbon Black's REST APIs, cbapi provides Pythonic APIs to make working with the Live Response API much easier.

In addition to easy-to-use APIs to call into Live Response, cbapi also provides a "job-based" interface that allows cbapi to intelligently schedule large numbers of concurrent Live Response sessions across multiple sensors. Your code can then be notified when the jobs are complete, returning the results of the job if it succeeded or the Exception if it failed.

4.6.1 Getting Started with Live Response

The cbapi Live Response API is built around establishing a `cbapi.response.live_response.LiveResponseSession` object from a `cbapi.response.models.Sensor` Model Object. Then you can call methods on the `LiveResponseSession` object to perform Live Response actions on the target host. These calls are synchronous, meaning that they will wait until the action is complete and a result is available, before returning back to your script. Here's an example:

```
>>> from cbapi.response import *
>>> cb = CbResponseAPI()
>>> sensor = cb.select(Sensor).where("hostname:WIN-IA9NQ1GN8OI").first()
>>> with sensor.lr_session() as session:
...     print(session.get_file(r"c:\test.txt"))

this is a test
```

Since the Live Response API is synchronous, the script will not continue until either the Live Response session is established and the file contents are retrieved, or an exception occurs (in this case, either a timeout error or an error reading the file).

As seen in the example above, the `.lr_session()` method is context-aware. CB Response has a limited number of concurrent Live Response session slots (by default, only ten). By wrapping the `.lr_session()` call within a `with` context, the session is automatically closed at the end of the block and frees that slot for another concurrent Live Response session in another script or user context.

A full listing of methods in the cbapi Live Response API is available in the documentation for the `cbapi.live_response_api.CbLRSessionBase` class.

4.6.2 Live Response Errors

There are four classes of errors that you will commonly encounter when working with the Live Response API:

- A `cbapi.errors.TimeoutError` is raised if a timeout is encountered when waiting for a response for a Live Response API request.
- A `cbapi.response.live_response_api.LiveResponseError` is raised if an error is returned during the execution of a Live Response command on an endpoint. The `LiveResponseError` includes detailed information about the error that occurred, including the exact error code that was returned from the endpoint and a textual description of the error.
- A `cbapi.errors.ApiError` is raised if you attempt to execute a command that is not supported by the sensor; for example, attempting to acquire a memory dump from a sensor running a pre-5.1 version of the agent will fail with an `ApiError` exception.
- A `cbapi.errors.ServerError` is raised if any other error occurs; for example, a 500 Internal Server Error is returned from the Live Response API.

4.6.3 Job-Based API

The basic Synchronous API described above in the Getting Started section works well for small tasks, targeting one sensor at a time. However, if you want to execute the same set of Live Response commands across a larger number of sensors, the cbapi provides a Job-Based Live Response API. The Job-Based Live Response API provides a straightforward API to submit Live Response jobs to a scheduler, schedule those Live Response jobs on individual endpoints concurrently, and return results and any errors back to you when the jobs complete. The Job-Based Live Response API is a natural fit with the Event-Based API to create IFTTT-style pipelines; if an event is received via the Event API, then perform Live Response actions on the affected endpoint via the Live Response Job-Based API.

The Job-Based API works by first defining a reusable "job" to perform on the endpoint. The Job is simply a class or function that takes a Live Response session object as input and performs a series of commands. Jobs can be as simple as retrieving a registry key, or as complex as collecting the Chrome browser history for any currently logged-in users.

Let's look at an example Job to retrieve a registry key. This example job is pulled from the `get_reg_autoruns.py` example script:

```
class GetRegistryValue(object):
    def __init__(self, registry_key):
        self.registry_key = registry_key

    def run(self, session):
        reg_info = session.get_registry_value(self.registry_key)
        return time.time(), session.sensor_id, self.registry_key, reg_info["value_data
↪"]
```

To submit this job, you instantiate an instance of a `GetRegistryValue` class with the registry key you want to pull back from the endpoint, and submit the `.run()` method to the Live Response Job API:

```
>>> job = GetRegistryValue(regmod_path)
>>> registry_job = cb.live_response.submit_job(job.run, sensor_id)
```

Your script resumes execution immediately after the call to `.submit_job()`. The job(s) that you've submitted will be executed in a set of background threads managed by cbapi.

4.7 CbAPI Changelog

4.7.1 CbAPI 1.5.0 - Released July 23, 2019

Updates

- **CB LiveOps**
  - Start new LiveQuery (LQ) runs
  - Fetch LQ results
  - View LQ run status
  - Filter on LQ results
- **PSC Org Key Management**
  - Added support for org key management within CBAPI
  - Credentials utility for org keys
  - PR #166, #169, #170

Examples

- LiveQuery - manage_run.py
- LiveQuery - run_search.py

4.7.2 CbAPI 1.4.5 - Released July 11, 2019

Updates

- **CB ThreatHunter**
  - Route updates for process search, feed management, watchlist management
  - Enforce org_key presence
  - Org-based process search
  - Org-based event search
  - Org-based tree queries
- Minor updates for Python3 Compatibility

Examples

- Updated CB TH Process Search Example
- Added process_guid to process_tree example for ThreatHunter

4.7.3 CbAPI 1.4.4 - Released July 3, 2019

Updates

- Carbon Black UBS Support PR #142
- CB Response - Fixing bulk update for Alerts to use v1 route
- Updates to use yaml safe_load #157

Examples

- Refactored Carbon Black ThreatHunter examples
- Added process_guid to process_tree example for ThreatHunter

4.7.4 CbAPI 1.4.3 - Released May 7, 2019

Updates

- CB ThreatHunter - Feed fixes #156
- CB Response - Change Alert model object to use v2 route #155
- CB Response - Only view active LR sessions #154
- Removing refs to VT alliance feeds #144

Examples

- CB Defense - Create list_events_with_cmdline_csv.py #152
- CB Defense - Updated import link to proper module #148

4.7.5 CbAPI 1.4.2 - Released March 27, 2019

This release introduces additional support for CB PSC's ThreatHunter APIs

- Threat Intelligence APIs

4.7.6 CbAPI 1.4.1 - Released January 10, 2019

- Bug fixes
- Adding to authorized error to make it clear that users should check API creds

4.7.7 CbAPI 1.4.0 - Released January 10, 2019

This release introduces support for CB PSC's ThreatHunter APIs

- Process, Tree, and Search are supported with more to come

4.7.8 CbAPI 1.3.6 - Released February 14, 2018

This release has one critical fix:

- Fix a fatal exception when connecting to CB Response 6.1.x servers

4.7.9 CbAPI 1.3.5 - Released February 2, 2018

This release includes bugfixes and contributions from the Carbon Black community.

All products:

- More Python 3 compatibility fixes.
- Fix the `wait_for_completion` and `wait_for_output` options in the Live Response `.create_process()` method. If `wait_for_completion` is True, the call to `.create_process()` will block until the remote process has exited. If `wait_for_output` is True, then `.create_process()` will additionally wait until the output of the remote process is ready and return that output to the caller. Setting `wait_for_output` to True automatically sets `wait_for_completion` to True as well.
- The `BaseAPI` constructor now takes three new optional keyword arguments to control the underlying connection pool: `pool_connections`, `pool_maxsize`, and `pool_block`. These arguments are sent to the underlying `HTTPAdapter` used when connecting to the Carbon Black server. For more information on these parameters, see the Python requests module API documentation for HTTPAdapter.

CB Defense:

- Date/time stamps in the Device model object are now represented as proper Python datetime objects, rather than integers.
- The `policy_operations.py` example script's "Replace Rule" command is fixed.
- Add the CB Live Response job-based API.
- Add a new example script `list_devices.py`

CB Response:

- The `Process` and `Binary` model objects now return None by default when a non-existent attribute is referenced, rather than throwing an exception.

- Fixes to `walk_children.py` example script.
- Fix exceptions in enumerating child processes, retrieving path and MD5sums from processes.
- Multiple `.where()` clauses can now be used in the `Sensor` model object.
- Workaround implemented for retrieving/managing more than 500 banned hashes.
- Alert bulk operations now work on batches of 500 alerts.
- `.flush_events()` method on `Sensor` model object no longer throws an exception on CB Response 6.x servers.
- `.restart_sensor()` method now available for `Sensor` model object.
- Fix `user_operations.py` example script to eliminate exception when adding a new user to an existing team.
- Add `.remove_team()` method on `User` model object.
- Automatically set `cb.legacy_5x_mode` query parameter for all Process queries whenever a legacy Solr core (from CB Response 5.x) is loaded.
- Added `.use_comprehensive_search()` method to enable the "comprehensive search" option on a Process query. See the CB Developer Network documentation on Comprehensive Search for more information on "comprehensive search".
- Add `.all_childprocs()`, `.all_modloads()`, `.all_filemods()`, `.all_regmods()`, `.all_crossprocs()`, and `.all_netconns()` methods to retrieve process events from all segments, rather than the current process segment. You can also use the special segment "0" to retrieve process events across all segments.
- Fix `cmdline_filters` in the `IngressFilter` model object.

CB Protection:

- Tamper Protection can now be set and cleared in the `Computer` model object.

4.7.10 CbAPI 1.3.4 - Released September 14, 2017

This release includes a critical security fix and small bugfixes.

Security fix:

- The underlying CbAPI connection class erroneously disabled hostname validation by default. This does *not* affect code that uses CbAPI through the public interfaces documented here; it only affects code that accesses the new `CbAPISessionAdapter` class directly. This class was introduced in version 1.3.3. Regardless, it is strongly recommended that all users currently using 1.3.3 upgrade to 1.3.4.

Bug fixes:

- Add rule filename parameter to CB Defense `policy_operations.py` script's `add-rule` command.
- Add support for `tamperProtectionActive` attribute to CB Protection's `Computer` object.
- Work around CB Response issue- the `/api/v1/sensor` route incorrectly returns an HTTP 500 if no sensors match the provided query. CbAPI now catches this exception and will instead return an empty set back to the caller.

4.7.11 CbAPI 1.3.3 - Released September 1, 2017

This release includes security improvements and bugfixes.

Security changes:

- CbAPI enforces the use of HTTPS when connecting to on-premise CB Response servers.
- CbAPI can optionally require TLSv1.2 when connecting to Carbon Black servers.
  - Note that some versions of Python and OpenSSL, notably the version of OpenSSL packaged with Mac OS X, do not support TLSv1.2. This will cause CbAPI to fail to connect to CB Response 6.1+ servers which require TLSv1.2 cipher suites.
  - A new command, `cbapi check-tls`, will report the TLS version supported by your platform.
  - To enforce the use of TLSv1.2 when connecting to a server, add `ssl_force_tls_1_2=True` to that server's credential profile.
- Add the ability to "pin" a specific server certificate to a credential profile.
  - You can now force TLS certificate verification on self-signed, on-premise installations of CB Response or Protection through the `ssl_cert_file` option in the credential profile.
  - To "pin" a server certificate, save the PEM-formatted server certificate to a file, and put the full path to that PEM file in the `ssl_cert_file` option of that server's credential profile.
  - When using this option with on-premise CB Response servers, you may also have to set `ssl_verify_hostname=False` as the hostname in the certificate generated at install time is `localhost` and will not match the server's hostname or IP address. This option will still validate that the server's certificate is valid and matches the copy in the `ssl_cert_file` option.

Changes for CB Protection:

- The API now sets the appropriate "GET" query fields when changing fields such as the `debugFlags` on the Computer object.
- The `.template` attribute on the Computer model object has been renamed `.templateComputer`.
- Remove AppCatalog and AppTemplate model objects.

Changes for CB Response:

- Added `.webui_link` property to CB Response Query objects.
- Added `ban_hash.py` example.

Bug Fixes:

- Error handling is improved on Python 3. Live Response auto-reconnect functionality is now fixed on Python 3 as a result.
- Workaround implemented for CB Response 6.1 where segment_ids are truncated on Alerts. The `.process` attribute on an Alert now ignores the `segment_id` and links to the first Process segment.
- Fixed issue with `Binary.signed` and `CbModLoadEvent.is_signed`.

4.7.12 CbAPI 1.3.2 - Released August 10, 2017

This release introduces the Policy API for CB Defense. A sample `policy_operations.py` script is now included in the `examples` directory for CB Defense.

Other changes:

- CB Response
  - Bugfixes to the `User` Model Object.
  - New `user_operations.py` example script to manage users & teams.
  - Additional `Team` Model Object to add/remove/modify user teams.
  - New `check_datasharing.py` example script to check if third party data sharing is enabled for binaries on any sensor groups.
  - Documentation fix for the `User` Model Object.
  - Fix to the `watchlist_operations.py` example script.

4.7.13 CbAPI 1.3.1 - Released August 3, 2017

This is a bugfix release with minor changes:

- CB Response
  - Add `partition_operations.py` script to demonstrate the use of the StoragePartition model object.
  - Fix errors when accessing the `.start` attribute of child processes.
  - Fix errors generated by the `walk_children.py` example script. The output has been changed as well to indicate the process lifetime, console UI link, and command lines.
  - Add an `.end` attribute to the Process model object. This attribute reports back either `None` if the process is still executing, or the last event time associated with the process if it has exited. See the `walk_children.py` script for an example of how to calculate process lifetime.
  - Fix errors when using the `.parents` attribute of a Process.
  - Add `wait_for_completion` flag to `create_process` Live Response method, and default to `True`. The `create_process` method will now wait for the target process to complete before returning.
- CB Defense
  - Add `wait_for_completion` flag to `create_process` Live Response method, and default to `True`. The `create_process` method will now wait for the target process to complete before returning.

4.7.14 CbAPI 1.3.0 - Released July 27, 2017

This release introduces the Live Response API for CB Defense. A sample `cblr_cli.py` script is now included in the `examples` directory for both CB Response and CB Defense.

Other changes:

- CB Protection
  - You can now create new `FileRule` and `Policy` model objects in cbapi.
- CB Response
  - Added `watchlist_exporter.py` and `watchlist_importer.py` scripts to the CB Response examples directory. These scripts allow you to export Watchlist data in a human- and machine-readable JSON format and then re-import them into another CB Response server.
  - The `Sensor` Model Object now uses the non-paginated (v1) API by default. This fixes any issues encountered when iterating over all the sensors and receiving duplicate and/or missing sensors.
  - Fix off-by-one error in `CbCrossProcess` object.

  - Fix issue iterating through `Process` Model Objects when accessing processes generated from a 5.2 server after upgrading to 6.1.
  - Reduce number of API requests required when accessing sibling information (parents, children, and siblings) from the `Process` Model Object.
  - Retrieve all events for a process when using `segment` ID of zero on a CB Response 6.1 server.
  - Behavior of `Process.children` attribute has changed:
    * Only one entry is present per child (before there were up to two; one for the spawn event, one for the terminate event)
    * The timestamp is derived from the start time of the process, not the timestamp from the spawn event. the two timestamps will be off by a few microseconds.
    * The old behavior is still available by using the `Process.childprocs` attribute instead. This incurs a performance penalty as another API call will have to be made to collect the childproc information.
  - `Binary` Model Object now returns False for `.is_signed` attribute if it is set to `(Unknown)`.
- Moved the `six` Python module into cbapi and removed the external dependency.

4.7.15 CbAPI 1.2.0 - Released June 22, 2017

This release introduces compatibility with our new product, CB Defense, as well as adding new Model Objects introduced in the CB Protection 8.0 APIs.

Other changes:

- CB Response
  - New method `synchronize()` added to the `Feed` Model Object
- Bug fixes and documentation improvements

4.7.16 CbAPI 1.1.1 - Released June 2, 2017

This release includes compatibility fixes for CB Response 6.1. Changes from 1.0.1 include:

- Substantial changes to the `Process` Model Object for CB Response 6.1. See details below.
- New `StoragePartition` Model Object to control Solr core loading/unloading in CB Response 6.1.
- New `IngressFilter` Model Object to control ingress filter settings in CB Response 6.1.
- Fix issues with `event_export.py` example script.
- Add `.all_events` property to the `Process` Model Object to expose a list of all events across all segments.
- Add example script to perform auto-banning based on watchlist hits from CB Event Forwarder S3 output files.
- Add bulk operations to the `ThreatReport` and `Alert` Query objects:
  - You can now call `.set_ignored()`, `.assign()`, and `.change_status()` on an `Alert` Query object to change the respective fields for every Alert that matches the query.
  - You can now call `.set_ignored()` on a `ThreatReport` Query object to set or clear the ignored flag for every ThreatReport that matches the query.

Changes to `Process` Model Object for CB Response 6.1

CB Response 6.1 uses a new way of recording process events that greatly increases the speed and scale of collection, allowing you to store and search data for more endpoints on the same hardware. Details on the new database format can be found on the Developer Network website at the Process API Changes for CB Response 6.0 page.

The `Process` Model Object traditionally referred to a single "segment" of events in the CB Response database. In CB Response versions prior to 6.0, a single segment will include up to 10,000 individual endpoint events, enough to handle over 95% of the typical event activity for a given process. Therefore, even though a `Process` Model Object technically refers to a single *segment* in a process, since most processes had less than 10,000 events and therefore were only comprised of a single segment, this distinction wasn't necessary.

However, now that processes are split across many segments, a better way of handling this is necessary. Therefore, CB Response 6.0 introduces the new `.group_by()` method. This method is new in cbapi 1.1.0 and is part of five new query filters available when communicating with a CB Response 6.1 server. These filters are accessible via methods on the `Process` Query object. These new methods are:

- `.group_by()` - Group the result set by a field in the response. Typically you will want to group by `id`, which will ensure that the result set only has one result per *process* rather than one result per *event segment*. For more information on processes, process segments, and how segments are stored in CB Response 6.0, see the Process API Changes for CB Response 6.0 page on the Developer Network website.
- `.min_last_update()` - Only return processes that have events after a given date/time stamp (relative to the individual sensor's clock)
- `.max_last_update()` - Only return processes that have events before a given date/time stamp (relative to the individual sensor's clock)
- `.min_last_server_update()` - Only return processes that have events after a given date/time stamp (relative to the CB Response server's clock)
- `.max_last_server_update()` - Only return processes that have events before a given date/time stamp (relative to the CB Response server's clock)

Examples for new Filters

Let's take a look at an example:

```
>>> from datetime import datetime, timedelta
>>> yesterday = datetime.utcnow() - timedelta(days=1)       # Get "yesterday" in GMT
>>> for proc in c.select(Process).where("process_name:cmd.exe").min_last_
↪update(yesterday):
...     print proc.id, proc.segment
DEBUG:cbapi.connection:HTTP GET /api/v1/process?cb.min_last_update=2017-05-21T18%3A41
↪%3A58Z&cb.urlver=1&facet=false&q=process_name%3Acmd.exe&rows=100&sort=last_
↪update+desc&start=0 took 2.164s (response 200)
00000001-0000-0e48-01d2-c2a397f4cfe0 1495465643405
00000001-0000-0e48-01d2-c2a397f4cfe0 1495465407157
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463680155
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463807694
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463543944
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463176570
00000001-0000-0e48-01d2-c2a397f4cfe0 1495463243492
```

Notice that the "same" process ID is returned seven times, but with seven different segment IDs. CB Response will return *every* process event segment that matches a given query, in this case, any event segment that contains the process command name `cmd.exe`.

That is, however, most likely not what you wanted. Instead, you'd like a list of the *unique* processes associated with the command name `cmd.exe`. Just add the `.group_by("id")` filter to your query:

```
>>> for proc in c.select(Process).where("process_name:cmd.exe").min_last_
↪update(yesterday).group_by("id"):
...     print proc.id, proc.segment
DEBUG:cbapi.connection:HTTP GET /api/v1/process?cb.group=id&cb.min_last_update=2017-
↪05-21T18%3A41%3A58Z&cb.urlver=1&facet=false&q=process_name%3Acmd.exe&rows=100&
↪sort=last_update+desc&start=0 took 2.163s (response 200)
00000001-0000-0e48-01d2-c2a397f4cfe0 1495465643405
```

CHAPTER 5

API Documentation

Once you've taken a look at the User Guide, read through some of the examples on GitHub, and maybe even written some code of your own, the API documentation can help you get the most out of cbapi by documenting all of the methods available to you.

5.1 CB Response API

This page documents the public interfaces exposed by cbapi when communicating with a Carbon Black Enterprise Response server.

5.1.1 Main Interface

To use cbapi with Carbon Black Response, you will be using the CbResponseAPI. The CbResponseAPI object then exposes two main methods to access data on the Carbon Black server: `select` and `create`.

5.1.2 Queries

5.1.3 Models

5.1.4 Live Response

File Operations

Registry Operations

Process Operations

5.2 CB Protection API

This page documents the public interfaces exposed by cbapi when communicating with a Carbon Black Enterprise Protection server.

5.2.1 Main Interface

To use cbapi with Carbon Black Protection, you will be using the CbProtectionAPI. The CbProtectionAPI object then exposes two main methods to select data on the Carbon Black server:

5.2.2 Queries

5.2.3 Models

5.3 CB Defense API

This page documents the public interfaces exposed by cbapi when communicating with a CB Defense server.

5.3.1 Main Interface

To use cbapi with Carbon Black Defense, you will be using the CBDefenseAPI. The CBDefenseAPI object then exposes two main methods to select data on the Carbon Black server:

5.3.2 Queries

5.3.3 Models

5.4 CB ThreatHunter API

This page documents the public interfaces exposed by cbapi when communicating with a Carbon Black PSC ThreatHunter server.

5.4.1 Main Interface

To use cbapi with Carbon Black ThreatHunter, you use CbThreatHunterAPI objects. These objects expose two main methods to access data on the ThreatHunter server: `select` and `create`.

5.4.2 Queries

The ThreatHunter API uses QueryBuilder instances to construct structured or unstructured (i.e., raw string) queries. You can either construct these instances manually, or allow `CbThreatHunterAPI.select()` to do it for you:

5.4.3 Models

5.5 CB LiveQuery API

This page documents the public interfaces exposed by cbapi when communicating with Carbon Black LiveQuery devices.

5.5.1 Main Interface

To use cbapi with Carbon Black LiveQuery, you use CbLiveQueryAPI objects.

The LiveQuery API is used in two stages: run submission and result retrieval.

5.5.2 Queries

The LiveQuery API uses QueryBuilder instances to construct structured or unstructured (i.e., raw string) queries. You can either construct these instances manually, or allow `CbLiveQueryAPI.select()` to do it for you:

5.5.3 Models

5.6 Exceptions

If an error occurs, the API attempts to roll the error into an appropriate Exception class.

5.6.1 Exception Classes

CHAPTER 6

Indices and tables

- genindex
- modindex
- search