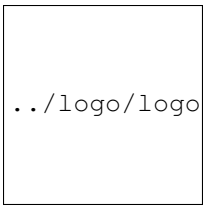
dbling Documentation

Release 1.0.0a

Mike Mabey

Sep 08, 2017

Contents



dbling is a tool for performing forensics in Chrome OS. More info will be forthcoming.

CHAPTER 1

Installation

Coming soon!

CHAPTER 2

dbling Components

dbling is divided into the following main components:

Crawler

The Crawler finds and downloads the list of the currently-available extensions on the Chrome Web Store, determines which extensions are at a version that has already been downloaded, downloads those that have not yet been downloaded, and adds information on the newly downloaded extensions to the database.

The code for the Crawler is under *crawl: The Chrome Web Store Crawler*.

Template Generator

The Template Generator runs concurrently with the Crawler. For each new extension downloaded by the Crawler, the Template Generator calculates the centroid of the extension and stores it in the database. The Template Generator does not run inside Chrome or Chrome OS, and so it does not use the same mechanisms for unpacking and installing that Chrome does natively. Instead, the primary function of the Template Generator is to mimic as closely as possible the Chrome's functions as they pertain to unpacking and installing extensions.

The code for the Template Generator is implemented alongside the Crawler, but the main function that creates templates is `calc_centroid()`.

Profiler

Coming soon!

MERL Exporter

Coming soon!

gripper

Coming soon!

CHAPTER 3

License

dbling is licensed under the MIT License.

dbling API

`crawl`: The Chrome Web Store Crawler

tasks

Tasks for Celery workers.

Beat Tasks

Beat tasks are those that are run on a periodic basis, depending on the configuration in `celeryconfig.py` or any cron jobs setup in the Ansible playbooks. Beat tasks only initiate the workflow by creating the jobs, they don't actually do the work for each task.

Entry Points

Entry points are where an actual worker begins its work. A single task corresponds to a specific CRX file. The task function dictates what operations are performed on the CRX. Each operation is represented by a specific worker function (as described below).

Worker Functions

Worker functions each represent a discrete action to be taken on a CRX file.

Helper Tasks and Functions

These functions provide additional functionality that don't fit in any of the above categories.

db_iface

webstore_iface

Chrome Web Store interface for dbling.

exception `crawl.webstore_iface.`**`ListDownloadFailedError`**(**args*, ***kwargs*)
Raised when the list download fails.

Initialize RequestException with `request` and `response` objects.

exception `crawl.webstore_iface.`**`ExtensionUnavailable`**
Raised when an extension isn't downloadable.

exception `crawl.webstore_iface.`**`BadDownloadURL`**
Raised when the ID is valid but we can't download the extension.

exception `crawl.webstore_iface.`**`VersionExtractError`**
Raised when extracting the version number from the URL fails.

class `crawl.webstore_iface.`**`DownloadCRXList`**(*ext_url*, ***, *return_count=False*, *session=None*)
Generate list of extension IDs downloaded from Google.

As a generator, this is designed to be used in a `for` loop. For example:

```
>>> crx_list = DownloadCRXList(download_url)
>>> for crx_id in crx_list:
...     print(crx_id)
```

The list of CRXs will be downloaded just prior to when the first item is generated. In other words, instantiating this class doesn't start the download, iterating over the instance starts the download. This is significant given that downloading the list is quite time consuming.

Parameters

- **`ext_url`** (*`str`*) – Specially crafted URL that will let us download the list of extensions.
- **`return_count`** (*`bool`*) – When True, will return a tuple of the form: `(crx_id, job_number)`, where `job_number` is the index of the ID plus 1. This way, the job number of the last ID returned will be the same as `len(DownloadCRXList)`.
- **`session`** (*`requests.Session`*) – Session object to use when downloading the list. If None, a new `requests.Session` object is created.

`download_ids`()
Starting point for downloading all CRX IDs.

This function actually creates an event loop and starts the downloads asynchronously.

Return type None

`_async_download_lists`()
Download, loop through the list of lists, combine IDs from each.

Return type None

`_dl_parse_id_list`(*list_url*)
Download the extension list at the given URL, return set of IDs.

Parameters **list_url** (*str*) – URL of an individual extension list.

Returns Set of CRX IDs.

Return type set

crawl.webstore_iface.**save_crx**(*crx_obj*, *download_url*, *save_path=None*, *session=None*)

Download the CRX, save in the save_path directory.

The saved file will have the format: <extension ID>_<version>.crx

If save_path isn't given, this will default to a directory called "downloads" in the CWD.

Adds the following keys to crx_obj:

- version: Version number of the extension, as obtained from the final URL of the download. This may differ from the version listed in the extension's manifest.
- filename: The basename of the CRX file (not the full path)
- full_path: The location (full path) of the downloaded CRX file

Parameters

- **crx_obj** (*munch.Munch*) – Previously collected information about the extension.
- **download_url** (*str*) – The URL template that already contains the correct Chrome version information and {} where the ID goes.
- **save_path** (*str or None*) – Directory where the CRX should be saved.
- **session** (*requests.Session or None*) – Optional Session object to use for HTTP requests.

Returns Updated version of crx_obj with version, filename, and full_path information added. If the download wasn't successful, not all of these may have been added, depending on when it failed.

Return type munch.Munch

merl: Matching Extension Ranking List Files

common: Modules Used Throughout dbling

centroid: Representation of a Centroid

clr: Color Text Easily

Color text.

Typical usage:

```
>>> red('red text', False)
```

Returns the string "red text" where the text will be red and the background will be the default.

```
>>> red('red background')
```

Returns the string "red background" where the text will be the default color and the background will be red.

common.clr.**add_color_log_levels**(*center=False*)
Alter log level names to be colored.

Levels are colored to have black text and a background colored as follows:

- Level 50 (Critical): red
- Level 40 (Error): magenta
- Level 30 (Warning): yellow
- Level 20 (Info): blue
- Level 10 (Debug): green
- Level 0 (Not Set): white

Parameters **center** (*bool*) – If log text should be centered. When set to `True`, the text will be centered to the width of `"CRITICAL"`, which is 8 characters. This makes it so the level in the log output always takes up the same number of characters.

Return type None

common.clr.**black**(*text*, *background=True*)
Set text (or its background) to be black.

common.clr.**red**(*text*, *background=True*)
Set text (or its background) to be red.

common.clr.**green**(*text*, *background=True*)
Set text (or its background) to be green.

common.clr.**yellow**(*text*, *background=True*)
Set text (or its background) to be yellow.

common.clr.**blue**(*text*, *background=True*)
Set text (or its background) to be blue.

common.clr.**magenta**(*text*, *background=True*)
Set text (or its background) to be magenta.

common.clr.**cyan**(*text*, *background=True*)
Set text (or its background) to be cyan.

common.clr.**white**(*text*, *background=True*)
Set text (or its background) to be white.

const: Constant Values

Constant values used by dbling.

common.const.**IN_PAT_VAULT = re.compile('^/?home/\\.shadow/[0-9a-z]*?/vault/user/')**
Regular expression pattern for including only the user's files

common.const.**ENC_PAT = re.compile('/ECRYPTFS_FNEK_ENCRYPTED\\.([^/]*)$')**
Regular expression pattern for identifying encrypted files

common.const.**SLICE_PAT = re.compile('.*(/home.*)')**

common.const.**CRX_URL = 'https://chrome.google.com/webstore/detail/%s'**
URL used for downloading CRXs

common.const.**ISO_TIME = '%Y-%m-%dT%H:%M:%SZ'**
ISO format for date time values

common.const.**DENTRY_FIELD_BYTES = 8**
Number of bytes used by the dir entry fields (preceding the filename)

class common.const.**FType**
File types as stored in directory entries in ext2, ext3, and ext4.

common.const.**MODE_UNIX = {32768: 1, 16384: 2, 24576: 4, 40960: 7, 4096: 5, 8192: 3, 49152: 6}**
Maps the octal values that `stat` returns from `stat.S_IFMT` to one of the regular Unix file types

common.const.**TYPE_TO_NAME = {0: '-', 1: 'r', 2: 'd', 3: 'c', 4: 'b', 5: 'p', 6: 's', 7: 'l'}**
Maps Unix file type numbers to the character used in DFXML to represent that file type

See: https://github.com/dfxml-working-group/dfxml_schema/blob/4c8aab566ea44d64313a5e559b1ecdce5348cecf/dfxml.xsd#L412

Other file types defined in DFXML schema

- h - Shadow inode (Solaris)
- w - Whiteout (OpenBSD)
- v - Special (Used in The SleuthKit for added "Virtual" files, e.g. $FAT1)

class common.const.**ModeTypeDT**
File types as stored in the file's mode.

In Linux, `fs.h` defines these values and stores them in bits 12-15 of `stat.st_mode`, e.g. `(i_mode >> 12) & 15`. In `fs.h`, the names are prefixed with `DT_`, hence the name of this enum class. Here are the original definitions:

```
#define DT_UNKNOWN      0
#define DT_FIFO         1
#define DT_CHR          2
#define DT_DIR          4
#define DT_BLK          6
#define DT_REG          8
#define DT_LNK          10
#define DT_SOCK         12
#define DT_WHT          14
```

common.const.**mode_to_unix**(*x*)
Return the UNIX version of the mode returned by `stat`.

common.const.**ECRYPTFS_SIZE_THRESHOLDS = (84, 104, 124, 148, 168, 188, 212, 232, 252, -inf)**
The index of these correspond with i such that 16*i is the lower bound and (16*(i+1))-1 is the upper bound for file name lengths that correspond to this value. Anything 16*9=144 or longer is invalid.

common.const.**ECRYPTFS_FILE_HEADER_BYTES = 8192**
Number of bytes used by eCryptfs for its header

common.const.**USED_FIELDS = ('_c_num_child_dirs', '_c_num_child_files', '_c_mode', '_c_depth', '_c_type')**
Fields used to calculate centroids

common.const.**USED_TO_DB = {'_c_num_child_dirs': 'num_dirs', '_c_type': 'type', '_c_num_child_files': 'num_files', '_c_**
Mapping of USED_FIELDS to database colulmn names. USED_TO_DB doesn't have the ttl_files field because it's not explicitly stored in the graph object.

graph: Customized Digraph Object

sync: Easy Mutex Creation

Context manager for easily using a pymemcache mutex.

The *acquire_lock* context manager makes it easy to use pymemcache (which uses memcached) to create a mutex for a certain portion of code. Of course, this requires the pymemcache library to be installed, which in turn requires memcached to be installed.

exception common.sync.**LockUnavailable**
Raised when a cached lock is already in use.

common.sync.**acquire_lock**(*lock_id*, *wait=0*, *max_retries=0*)
Acquire a lock on the given lock ID, or raise an exception.

This context manager can be used as a mutex by doing something like the following:

```
>>> from time import sleep
>>> job_done = False
>>> while not job_done:
...     try:
...         with acquire_lock('some id'):
...             sensitive_function()
...             job_done = True
...     except LockUnavailable:
...         # Sleep for a couple seconds while the other code runs and
...         # hopefully completes
...         sleep(2)
```

In the above example, `sensitive_function()` should only be run if no other code is also running it. A more concise way of writing the above example would be to use the other parameters, like this:

```
>>> with acquire_lock('some id', wait=2):
...     sensitive_function()
```

Parameters

- **lock_id** (*str or bytes*) – The ID for this lock. See pymemcache's documentation on key constraints for more info.
- **wait** (*int*) – Indicates how many seconds after failing to acquire the lock to wait (sleep) before retrying. When set to 0 (default), will immediately raise a *LockUnavailable* exception.
- **max_retries** (*int*) – Maximum number of times to retry to acquire the lock before raising a *LockUnavailable* exception. When set to 0 (default), will always retry. Has essentially no effect if `wait` is 0.

Raises ***LockUnavailable*** – when a lock with the same ID already exists and `wait` is set to 0.

util: Various Utilities for dbling

common.util.**validate_crx_id**(*crx_id*)
Validate the given CRX ID.

Check that the Chrome extension ID has three important properties:

1. It must be a string

2. It must have alpha characters only (strictly speaking, these should be lowercase and only from `a-p`, but checking for this is a little overboard)
3. It must be 32 characters long

Parameters **crx_id** (*str*) – The ID to validate.

Raises **MalformedExtID** – When the ID doesn't meet the criteria listed above.

exception common.util.**MalformedExtId**
Raised when an ID doesn't have the correct form.

common.util.**get_crx_version**(*crx_path*)
Extract and return the version number from the CRX's path.

The return value from the download() function is in the form: `<extension ID>_<version>.crx`.

The `<version>` part of that format is "x_y_z" for version "x.y.z". To convert to the latter, we need to 1) get the basename of the path, 2) take off the trailing ".crx", 3) remove the extension ID and '_' after it, and 4) replace all occurrences of '_' with '.'.

Parameters **crx_path** (*str*) – The full path to the downloaded CRX, as returned by the download() function.

Returns The version number in the form "x.y.z".

Return type str

common.util.**get_id_version**(*crx_path*)
From the path to a CRX, extract and return the ID and version as strings.

Parameters **crx_path** (*str*) – The full path to the downloaded CRX.

Returns The ID and version number as a tuple: `(id, num)`

Return type tuple(str, str)

common.util.**separate_mode_type**(*mode*)
Separate out the values for the mode (permissions) and the file type from the given mode.

Both returned values are integers. The mode is just the permissions (usually displayed in the octal format), and the type corresponds to the standard VFS types:

- 0: Unknown file
- 1: Regular file
- 2: Directory
- 3: Character device
- 4: Block device
- 5: Named pipe (identified by the Python stat library as a FIFO)
- 6: Socket
- 7: Symbolic link

Parameters **mode** (*int*) – The mode value to be separated.

Returns Tuple of ints in the form: `(mode, type)`

Return type tuple(int, int)

common.util.**calc_chrome_version**(*last_version*, *release_date*, *release_period=10*)
Calculate the most likely version number of Chrome.

The calculation is based on the last known version number and its release date, based on the number of weeks (release_period) it usually takes to release the next major version. A list of releases and their dates is available on Wikipedia.

Parameters

- **last_version** (*str*) – Last known version number, e.g. "43.0". Should only have the major and minor version numbers and exclude the build and patch numbers.
- **release_date** (*list*) – Release date of the last known version number. Must be a list of three integers: [YYYY, MM, DD].
- **release_period** (*int*) – Typical number of weeks between releases.

Returns The most likely current version number of Chrome in the same format required of the last_version parameter.

Return type str

common.util.**make_download_headers**()
Return a dict of headers to use when downloading a CRX.

Returns Set of HTTP headers as a dict, where the key is the header type and the value is the header content.

Return type dict[str, str]

common.util.**dt_dict_now**()
Return a dict of the current time.

Returns

A dict with the following keys:

- year
- month
- day
- hour
- minute
- second
- microsecond

Return type dict[str, int]

common.util.**dict_to_dt**(*dt_dict*)
Reverse of *dt_dict_now()*.

Parameters **dt_dict** (*dict*) – A dict (such as *dt_dict_now()* returns) that correspond with the keyword parameters of the datetime constructor.

Returns A datetime object.

Return type datetime.datetime

class common.util.**MunchyMunch**(*f*)
Wrapper class to munchify crx_obj parameters.

This wrapper converts either the kwarg `crx_obj` or the first positional argument (tests in that order) to a `Munch` object, which allows us to refer to keys in the `Munch` dictionary as if they were attributes. See the docs on the `munch` library for more information.

Example usage:

```
>>> @MunchyMunch
... def test_func(crx_obj)
...     # crx_obj will be converted to a Munch
...     print(crx_obj.id)
```

Parameters **f** – The function to wrap.

common.util.**byte_len**(*s*)

Return the length of `s` in number of bytes.

Parameters **s** (*str or bytes*) – The `string` or `bytes` object to test.

Returns The length of `s` in bytes.

Return type int

Raises **TypeError** – If `s` is not a `str` or `bytes`.

common.util.**ttl_files_in_dir**(*dir_path*, *pat='.'*)

Count the files in the given directory.

Will count all files except `.` and `..`, including any files whose names begin with `.` (using the `-A` option of `ls`).

Parameters

- **dir_path** (*str*) – Path to the directory.
- **pat** (*str*) – Pattern the files should match when searching. This is passed to `grep`, so when the default remains (`.`), it will match all files and thus not filter out anything.

Returns The number of files in the directory.

Return type int

Raises **NotADirectoryError** – When `dir_path` is not a directory.

common.util.**chunkify**(*iterable*, *chunk_size*)

Split an iterable into smaller iterables of a certain size (chunk size).

For example, say you have a list that, for whatever reason, you don't want to process all at once. You can use *chunkify()* to easily split up the list to whatever size of chunk you want. Here's an example of what this might look like:

```
>>> my_list = range(1, 6)
>>> for sublist in chunkify(my_list, 2):
...     for i in sublist:
...         print(i, end=', ')
...     print()
```

The output of the above code would be:

```
1, 2,
3, 4,
5,
```

Idea borrowed from http://code.activestate.com/recipes/303279-getting-items-in-batches/.

Parameters

- **iterable** – The iterable to be split into chunks.
- **chunk_size** (*int*) – Size of each chunk. See above for an example.

Secret Files

The `secret` directory is used to store sensitive information specific to an installation of dbling. The files in this directory have excluded from the repository for obvious reasons, but should include a `creds.py` file. It has should have a form such as displayed below.

```
import yaml
from os import uname
from os.path import join, dirname

with open(join(dirname(__file__), 'passes.yml')) as passes:
    passwd = yaml.load(passes)

crx_save_path = ''  # Path where the CRXs should be saved when downloaded
db_info = {  # Database access information
    'uri': '',  # Full URI for accessing the DB. See SQLAlchemy docs for more info.
    'user': '',
    'pass': '',
    'nodes': ['host1', ],  # Host names of machines that should use 127.0.0.1 instead␣
↪of the value for full_url below
    'full_url': '1.2.3.4',  # IP address of host with the database (usually dbling␣
↪master)
}
# Login info for workers to access the celery server on the dbling master
celery_login = {'user': 'sample_username', 'pass': 'secure_password', 'port': 5672}
admin_emails = (  # Names and email addresses of admins that should receive emails␣
↪from Celery
    ('Admin Name', 'admin_email@example.com'),
)
sender_email_addr = 'ubuntu@{}'.format(uname().nodename)  # Email address Celery␣
↪should use when sending admin emails
```

The template above references another file that should be in the `secret` directory, `passes.yml`. This should have a form as shown below. Without this file, the Ansible playbooks will not function properly.

```
---

mysql_rt_pass: ''  # MySQL root user password
mysql_dbling_user: 'dbling_dbusr'  # MySQL regular user name
mysql_dbling_pass: ''  # MySQL regular user password

rabbit_user: 'dbling_crawler'  # RabbitMQ user name
rabbit_pass: ''  # RabbitMQ user password
```

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

I

L

M

R

S

T

U

V

W

Y