Creating a mac_apt Plugin (Part 2)

In IIR Vol.54, we took a look at the demo plugin provided by the mac_apt forensic analysis framework for macOS to understand the basic structure of mac_apt plugins[*1]. In this installment, I discuss the data stored in "~/Library/Caches/<Application Bundle ID>/Cache.db" and go over the implementation of a mac_apt plugin for analyzing this artifact. If you haven't read the article in IIR Vol.54 yet, you may find it easier to follow along if you go back and read that first.

2.1 What Information Does Cache.db Hold?

First, we look at the information recorded in Cache.db. This is a cache of HTTP/HTTPS data transfers made via APIs like NSURLRequest. The Cache.db file is an SQLite-format database and holds the data in the database tables shown in Tables 1–5. The cache data is basically stored in this database, but data above a certain size is stored as a file in the fsCacheData directory (Figure 1). This is apparently called the "CFURL Cache", because the database table name starts with "cfurl_cache".

Column	Data type
entry_ID	The entry ID
response_object	BLOB in plist format. Holds the URL accessed, HTTP status, response header, etc.
request_object	BLOB in plist format. Holds the URL accessed, access method, request header, etc.
proto_props	Unknown (used for cache control?)
user_info	Unknown (all NULL to the extent this author has checked)

Table 1: cfurl_cache_blob_data

Column	Data type
entry_ID	The entry ID
isDataOnFS	Flag indicating the format of the response data from the server 0: Response body from the server is stored in receiver_data. 1: Response body from the server is stored in a file.
receiver_data	Holds the response data from the server (isDataOnFS = 0) or the name of the file in which it has been stored (isDataOnFS = 1). The file is saved in "~/Library/Caches/<App Bundle ID>/fsCacheData/". File names are in UUID format.

Table 2: cfurl_cache_receiver_data

Column	Data type
entry_ID	The entry ID
version	Unknown (all 0 to the extent this author has checked)
hash_value	Unknown
storage_policy	Unknown (all 0 to the extent this author has checked)
request_key	URL of the destination accessed
time_stamp	Timestamp of when the URL was accessed
partition	Unknown (all NULL to the extent this author has checked)

Table 3: cfurl_cache_response

Column	Data type
schema_version	The schema version (all 202 to the extent this author has checked, except in cases where the table itself does not exist)

Table 4: cfurl_cache_schema_version

Column	Data type
cfurl_cache_response	Maximum value for entry_ID of cfurl_cache_response?

Table 5: sqlite_sequence

*1 Focused Research (1) in Internet Infrastructure Review (IIR) Vol.54, "Creating a mac_apt Plugin (Part 1)" (https://www.iij.ad.jp/en/dev/iir/054.html).

Looking at this information, you can check not only the date and time of program data transfers and the destination URLs but also the responses received from the servers. User and program activity histories are crucial to forensic analysis, so this artifact is a very useful source of information.

2.2 Designing a Plugin

2.2.1 Data to Acquire

Before we get into creating the plugin, let's look a little closer at what information we can get from the artifact.

Tables 1–3 indicate we can obtain the data transfer timestamp, the destination URL, the client's request method and header, the server's HTTP status and response body, and the response body from the server. The data in these tables are linked by a key called entry_ID. And the "<Application Bundle ID>" part of the file path to where Cache.db is stored identifies the program that made the transfer. We need the plugin to collect this information and save it as an analysis result.

Note that the response_object and request_object in the cfurl_cache_blob_data table (Table 1) are stored as plist-format BLOBs (Figure 2)[*2]. This data should not be left in plist format when stored in the analysis results. Instead, it should be parsed to make it easy for the analyst to determine the contents.

```
% ls -alR ~/Library/Caches/com.apple.osascript
total 128
drwxr-xr-x    4 macforensics  staff    128  2 10 17:36 .
drwx------+ 150 macforensics  staff   4800  4 27 15:12 ..
-rw-r--r--@   1 macforensics  staff  65536  5 19  2021 Cache.db
drwxr-xr-x@   4 macforensics  staff    128  2  2  2021 fsCachedData

/Users/macforensics/Library/Caches/com.apple.osascript/fsCachedData:
total 344
drwxr-xr-x@ 4 macforensics  staff     128  2  2  2021 .
drwxr-xr-x  4 macforensics  staff     128  2 10 17:36 ..
-rw-r--r--@ 1 macforensics  staff  116503 11  9  2020 2B1680C0-DAE0-4EA0-9EC0-C4FC7F86A8C0
-rw-r--r--@ 1 macforensics  staff   53755  2  2  2021 A391D5EC-9FCF-4993-A0AF-EEF2C871EF6A
```

Figure 1: File Structure

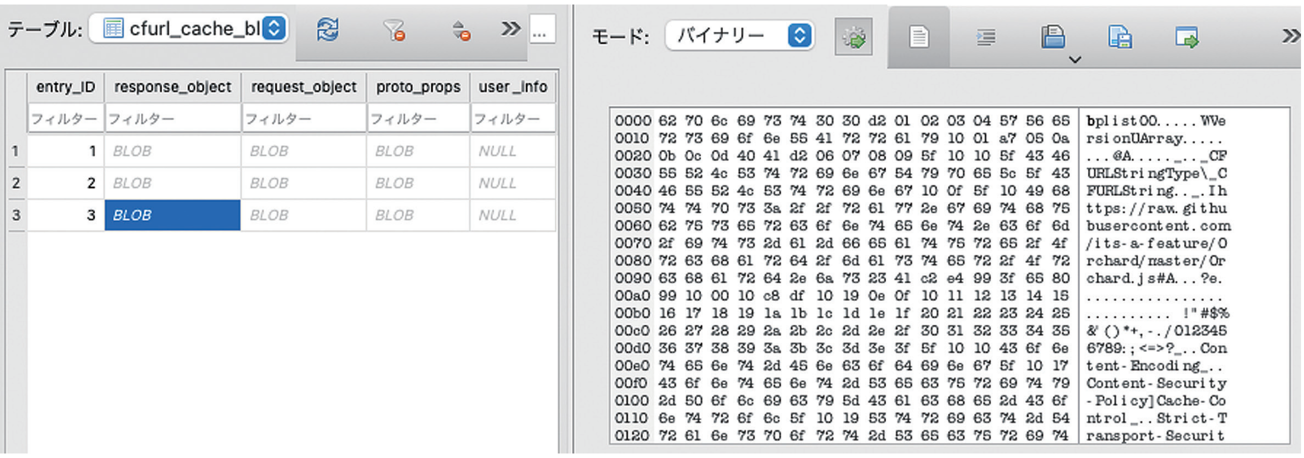

Figure 2: Sample response_object

*2 The "bplist00" sequence at the start is the plist binary format's magic number.

Figure 3 shows the result of exporting the request_object data using DB Browser for SQLite[*3] or other software, parsing it with plutil, the standard macOS command. It looks like the data we need for forensic analysis is contained in elements 18 and 19 of the Array. Element 18 is the HTTP request method, and 19 is the HTTP request header.

Element 19 also holds Base64-encoded data in its "__hhaa__" field. This is a plist in binary format; Figure 4 shows the decoded content. It is the same as the HTTP request header and thus we can conclude that it does not need to be included in the analysis results. The response_object data can also be examined in the same manner (Figure 5). In this case, element 3 of the Array holds the HTTP status and element 4 holds the HTTP response header.

```
% plutil -p cfurl_cache_blob_data__request_object_2.bin
{
  "Array" => [
    0 => 0
    1 => {
      "_CFURLString" => "https://www.example.com/"
      "_CFURLStringType" => 15
    }
    2 => 60
    3 => 1
    4 => "__CFURLRequestNullTokenString__"
    5 => 1
    6 => 134
    7 => "__CFURLRequestNullTokenString__"
    8 => "__CFURLRequestNullTokenString__"
    9 => 1
    10 => 0
    11 => 0
    12 => 0
    13 => 0
    14 => 0
    15 => -1
    16 => "__CFURLRequestNullTokenString__"
    17 => 2
    18 => "GET"
    19 => {
      "__hhaa__" => "

YnBsaXN0MDDTAQIDBAYIXxAPQWNjZXB0LUVuY29kaW5nVkFjY2VwdF8QD0FjY2VwdC1
MYW5ndWFnZaEFXxARZ3ppcCwgZGVmbGF0ZSwgYnKhB1MqLyqhCVVqYS1qcAgPISg6PF
BSVlgAAAAAAAABAQAAAAAAAAAKAAAAAAAAAAAAAAAAAAAAXg=="
      "Accept" => "*/*"
      "Accept-Encoding" => "gzip, deflate, br"
      "Accept-Language" => "ja-jp"
    }
    20 => "__CFURLRequestNullTokenString__"
    21 => "__CFURLRequestNullTokenString__"
  ]
  "Version" => 9
}
```

Figure 3: Result of Parsing the request_object Data

```
% echo
YnBsaXN0MDDTAQIDBAYIXxAPQWNjZXB0LUVuY29kaW5nVkFjY2VwdF8QD0FjY2VwdC1
MYW5ndWFnZaEFXxARZ3ppcCwgZGVmbGF0ZSwgYnKhB1MqLyqhCVVqYS1qcAgPISg6PF
BSVlgAAAAAAAABAQAAAAAAAAAKAAAAAAAAAAAAAAAAAAAAXg== | base64 -d |
plutil -p -
{
  "Accept" => [
    0 => "*/*"
  ]
  "Accept-Encoding" => [
    0 => "gzip, deflate, br"
  ]
  "Accept-Language" => [
    0 => "ja-jp"
  ]
}
```

Figure 4: Result of Parsing the request_object's __hhaa__ Field

```
% plutil -p cfurl_cache_blob_data__response_object_2.bin
{
  "Array" => [
    0 => {
      "_CFURLString" => "https://www.example.com/"
      "_CFURLStringType" => 15
    }
    1 => 628074307.809312
    2 => 0
    3 => 200
    4 => {
      "__hhaa__" => "

YnBsaXN0MDDdAQIDBAUGBwgJCgsMDQ4QEhQWGBocHiAiJCZcQ29udGVudC1UeXBlVEV
0YWdXWC1DYWNoZVNBZ2VfEBBDb250ZW50LUVuY29kaW5nVlNlcnZlcldFeHBpcmVzXU
NhY2hlLUNvbnRyb2xURGF0ZV5Db250ZW50LUxlbmd0aF1BY2NlcHQtUmFuZ2VzVFZhc
nldTGFzdC1Nb2RpZmllZKEPXxAYdGV4dC9odG1sOyBjaGFyc2V0PVVURi04oRFcIjMx
NDc1MjY5NDcioRNTSElUoRVWNTY4Nzk3oRdUZ3ppcKEZXkVDUyAobnliLzFEMkYpoRt
fEB1UaHUsIDAzIERlYyAyMDIwIDA5OjA1OjA3IEdNVKEdXm1heC1hZ2U9NjA0ODAwoR
9fEB1UaHUsIDI2IE5vdiAyMDIwIDA5OjA1OjA3IEdNVKEhUzY0OKEjVWJ5dGVzoSVfE
A9BY2NlcHQtRW5jb2RpbmehJ18QHVRodSwgMTcgT2N0IDIwMTkgMDc6MTg6MjYgR01U
AAgAIwAwADUAPQBBAFQAWwBjAHEAdgCFAJMAmACmAKgAwwDFANIA1ADYANoA4QDjAOg
A6gD5APsBGwEdASwBLgFOAVABVAFWAVwBXgFwAXIAAAAAAAACAQAAAAAAAAAoAAAAAA
AAAAAAAAAAAABkg=="
      "Accept-Ranges" => "bytes"
      "Age" => "568797"
      "Cache-Control" => "max-age=604800"
      "Content-Encoding" => "gzip"
      "Content-Length" => "648"
      "Content-Type" => "text/html; charset=UTF-8"
      "Date" => "Thu, 26 Nov 2020 09:05:07 GMT"
      "Etag" => ""3147526947""
      "Expires" => "Thu, 03 Dec 2020 09:05:07 GMT"
      "Last-Modified" => "Thu, 17 Oct 2019 07:18:26 GMT"
      "Server" => "ECS (nyb/1D2F)"
      "Vary" => "Accept-Encoding"
      "X-Cache" => "HIT"
    }
    5 => 1256
    6 => "text/html"
  ]
  "Version" => 1
}
```

Figure 5: Result of Parsing the response_object Data

*3 DB Browser for SQLite (https://sqlitebrowser.org/).

The receiver_data field in the cfurl_cache_receiver_data table (Table 2) holds the response body received from the server. This information is also useful for forensic analysis, so we save it in the analysis results. But rather than including the data in the analysis results in the form of a file saved in the fsCacheData directory, we should instead export the file. This avoids unnecessarily increasing the analysis results file size.

2.2.2 Data Acquisition Method

Next, we consider how to go about acquiring the data. Since Cache.db is a SQLite database, information can be retrieved using SQL queries. Given the results above, we can use the SQL query in Figure 6 to obtain the required information.

The plist data stored in the response_object and request_object can be parsed using the Python plistlib module[*4]. __hhaa__ is excluded from the parse result.

2.2.3 Plugin Implementation Approach

At this point, we have determined what data to obtain and how to acquire it, and based on this, we can lay out an implementation approach for the plugin as shown below.

1. Process a macOS disk image or exported artifacts.
   1.1 If processing a disk image, process all artifacts in all users' "~/Library/Caches/" directories.
   1.2 If processing exported artifacts, process all artifacts in the specified directory.
2. Save the following data in the analysis results.
   2.1 Data obtained via the SQL query in Figure 6
   2.2 Results of parsing response_object and request_object
   2.3 Application Bundle ID
   2.4 In the case of disk images, export the files in the fsCacheData directory

```
SELECT entry_ID, time_stamp, request_key, request_object, response_object, isDataOnFS, receiver_data
       FROM cfurl_cache_response JOIN cfurl_cache_blob_data USING (entry_ID)
       JOIN cfurl_cache_receiver_data USING (entry_ID)
```

Figure 6: SQL Query for Obtaining the Required Data from Cache.db

*4 You can also use the CommonFunctions.ReadPlist() method provided by mac_apt.

2.3 Creating the Plugin

Now let's start creating the plugin. The plugin discussed here was merged via a Pull Request in July 2021, so you can find it on the mac_apt GitHub repository[*5].

Here, I give an outline of what the plugin does. Please refer to the source code for further details if necessary[*6].

2.3.1 Properties

I set the properties as shown in Figure 7. The plugin is called CFURLCACHE. As it processes disk images and exported artifacts, it uses "__Plugin_Modes = "MACOS,ARTIFACTONLY"".

2.3.2 Entry Points

■ Plugin_Start()

Plugin_Start () is the plugin entry point when you run mac_apt.py (Figure 8).

mac_info.users holds a list object containing the user information on the disk image. You can use this to loop through and process the artifacts for all users (line 179). But since the same home directory can be configured for use on multiple accounts, the plugin skips home directories that have already been processed (lines 180–182).

```
__Plugin_Name = "CFURLCACHE" # Cannot have spaces, and must be all caps!
__Plugin_Friendly_Name = "CFURL cache"
__Plugin_Version = "1.0"
__Plugin_Description = "Parses CFURL cache and extract date, URL, request, response, and received data."
__Plugin_Author = "Minoru Kobayashi"
__Plugin_Author_Email = "unknownbit@gmail.com"

__Plugin_Modes = "MACOS,ARTIFACTONLY" # Valid values are 'MACOS', 'IOS, 'ARTIFACTONLY'
__Plugin_ArtifactOnly_Usage = 'Provide the path to "/Library/Cache/" folder under user home'
```

Figure 7: Properties

```
173 def Plugin_Start(mac_info):
174     '''Main Entry point function for plugin'''
175     cfurl_cache_artifacts = []
176     cfurl_cache_base_path = '{}/Library/Caches/'
177     processed_paths = set()
178 
179     for user in mac_info.users:
180         if user.home_dir in processed_paths:
181             continue # Avoid processing same folder twice (some users have same folder! (Eg: root & daemon))
182         processed_paths.add(user.home_dir)
183         base_path = cfurl_cache_base_path.format(user.home_dir)
184         if not mac_info.IsValidFolderPath(base_path):
185             continue
186         cache_folder_list = mac_info.ListItemsInFolder(base_path, EntryType.FOLDERS, include_dates=False)
187         app_bundle_ids = [folder_item['name'] for folder_item in cache_folder_list]
188         for app_bundle_id in app_bundle_ids:
189             cache_folder_path = os.path.join(cfurl_cache_base_path.format(user.home_dir), app_bundle_id)
190             cache_db_path = os.path.join(cache_folder_path, 'Cache.db')
191             if mac_info.IsValidFilePath(cache_db_path) and mac_info.GetFileSize(cache_db_path) > 0:
192                 ExtractAndReadCFURLCache(mac_info, cfurl_cache_artifacts, user.user_name, app_bundle_id, cache_folder_path)
193 
194     if len(cfurl_cache_artifacts) > 0:
195         PrintAll(cfurl_cache_artifacts, mac_info.output_params, '')
196     else:
197         log.info('No CFURL cache artifacts were found!')
```

Figure 8: The Plugin_Start() Function

[*5] cfurl_cache.py (https://github.com/ydkhatri/mac_apt/blob/3e823ee36bdf133c4de3503848435033ee20943d/plugins/cfurl_cache.py).

[*6] Note that, if the plugin has been updated since this writing, line numbers in this article may not correspond to those in the current source code.

For unprocessed home directories, the mac_info object's IsValidFolderPath() method is used to check for the existence of the home directory in the disk image (line 184), and the ListItemsInFolder() method is used to create a list of directories in "~/Library/Caches/" (line 186).

The IsValidFilePath() method is then used to check for the existence of Cache.db files in each of the directories (line 191). If the file exists, the ExtractAndReadCFURLCache() function is called to obtain analysis results for Cache.db and export the artifact file (line 192). The analysis results are stored in cfurl_cache_artifacts as a list object.

Once all of the Cache.db files have been analyzed, the results are stored using the PrintAll() function (line 195).

■ Plugin_Start_Standalone()

Plugin_Start_Standalone() is the plugin entry point when you run mac_apt_artifact_only.py (Figure 9). input_files_list contains the name of the directory to be processed as specified on the command line (line 202). Beyond that, the function basically runs through the same process as Plugin_Start(), but with this entry point, the OpenAndReadCFURLCache() function is called to perform the analysis instead of the ExtractAndReadCFURLCache() function (line 212).

2.3.3 Data Analysis

■ ExtractAndReadCFURLCache()

This function opens a Cache.db file in the disk image, analyzes the data, saves the analysis results, and exports the artifact file (Figure 10).

Cache.db is opened inside the OpenDbFromImage() function (line 147). This function uses the connect() method of the SqliteWrapper class provided by mac_apt to get a connection to the SQLite database. As this class is a wrapper around the standard Python sqlite3

```
199 def Plugin_Start_Standalone(input_files_list, output_params):
200     '''Main entry point function when used on single artifacts (mac_apt_singleplugin), not on a full disk image'''
201     log.info("Module Started as standalone")
202     for input_path in input_files_list:
203         log.debug("Input file passed was: " + input_path)
204         cfurl_cache_artifacts = []
205         if os.path.isdir(input_path):
206             cache_folder_list = os.listdir(input_path)
207             app_bundle_ids = [f for f in cache_folder_list if os.path.isdir(os.path.join(input_path, f))]
208             for app_bundle_id in app_bundle_ids:
209                 cache_folder_path = os.path.join(input_path, app_bundle_id)
210                 cache_db_path = os.path.join(cache_folder_path, 'Cache.db')
211                 if os.path.isfile(cache_db_path) and os.path.getsize(cache_db_path) > 0:
212                     OpenAndReadCFURLCache(cfurl_cache_artifacts, '', app_bundle_id, cache_folder_path)
213 
214         if len(cfurl_cache_artifacts) > 0:
215             PrintAll(cfurl_cache_artifacts, output_params, input_path)
216         else:
217             log.info('No CFURL cache artifacts were found!')
```

Figure 9: The Plugin_Start_Standalone() Function

```
145 def ExtractAndReadCFURLCache(mac_info, cfurl_cache_artifacts, username, app_bundle_id, folder_path):
146     cfurl_cache_db_path = os.path.join(folder_path, 'Cache.db')
147     db, wrapper = OpenDbFromImage(mac_info, cfurl_cache_db_path, username)
148     if db:
149         ParseCFURLEntry(db, cfurl_cache_artifacts, username, app_bundle_id, cfurl_cache_db_path)
150         mac_info.ExportFolder(folder_path, os.path.join(__Plugin_Name, username), True)
151         db.close()
```

Figure 10: The ExtractAndReadCFURLCache() Function

module, you can use sqlite3 methods to execute SQL queries and so forth.

The data analysis happens not in the ExtractAndReadCFURLCache() function but in the ParseCFURLEntry() function. It is set up this way so that both ExtractAndReadCFURLCache() and OpenAndReadCFURLCache(), described below, can use the same processing routine (line 149).

Finally, the ExportFolder() method is used to export the artifact file (line 150). The first argument is the folder path to export from, the second is the name of the export destination folder, and the third is the overwrite flag.

The export destination folder specified by the second argument is created in the "Export" folder that is created within the output destination folder specified on the mac_apt command line. A look at the source code of other plugins shows that they basically use "__Plugin_Name" for this. But with CFURL Cache, because there is an artifact file for each user, the user name is also included in the export destination folder name.

■ OpenAndReadCFURLCache()

This function opens the exported Cache.db file, analyzes the data, and saves the analysis results (Figure 11). It does not export artifact files.

Cache.db is opened inside the OpenDb() function. This function gets a connection using the mac_apt CommonFunctions class's open_sqlite_db_readonly() method. Data analysis is done by the ParseCFURLEntry() function, as noted above.

■ ParseCFURLEntry()

This function issues a SQL query and retrieves the required data from Cache.db (Figure 12). It also parses the acquired data and saves the analysis results.

First, it gets a list of table names, and if a cfurl_cache_schema_version table exists, it gets the schema version (lines 118–121). To the extent I have checked, only version 202 is ever used[*7].

Next, it uses the SQL query in Figure 6 to get the required data (lines 125–128). As mentioned above, the request_object and response_object data are in a binary-format

```
def OpenAndReadCFURLCache(cfurl_cache_artifacts, username, app_bundle_id, folder_path):
    cfurl_cache_db_path = os.path.join(folder_path, 'Cache.db')
    db = OpenDb(cfurl_cache_db_path)
    if db:
        ParseCFURLEntry(db, cfurl_cache_artifacts, 'N/A', app_bundle_id, cfurl_cache_db_path)
        db.close()
```

Figure 11: The OpenAndReadCFURLCache() Function

*7 In some cases, depending on the macOS version, there is no cfurl_cache_schema_version table present.

plist. The data are analyzed by the ParseRequestObject() and ParseResponseObject() functions described below (lines 130–131).

The receiver_data object type depends on what it holds. If it holds the response body, it will be “bytes”. If it holds the name of the file (UUID) in the fsCacheData directory, it will be “str” (lines 132–135).

Finally, it saves the acquired data as an entry in the analysis results (lines 140–143). The plugin defines the CfurlCacheItem class to hold analysis results (Figure 13), and the entry is an instance of that class. The class has no methods; it is simply there to group the data together.

```
114 def ParseCFURLEntry(db, cfurl_cache_artifacts, username, app_bundle_id, cfurl_cache_db_path):
115     db.row_factory = sqlite3.Row
116     tables = CommonFunctions.GetTableNames(db)
117     schema_version = 0
118     if 'cfurl_cache_schema_version' in tables:
119         schema_version = CheckSchemaVersion(db)
120     else:
121         log.debug('There is no cfurl_cache_schema_version table.')
122 
123     if 'cfurl_cache_response' in tables:
124         if schema_version in (0, 202):
125             query = """SELECT entry_ID, time_stamp, request_key, request_object, response_object, isDataOnFS, receiver_data
126                     FROM cfurl_cache_response JOIN cfurl_cache_blob_data USING (entry_ID)
127                     JOIN cfurl_cache_receiver_data USING (entry_ID)"""
128             cursor = db.execute(query)
129             for row in cursor:
130                 http_req_method, req_headers = ParseRequestObject(row['request_object'])
131                 http_status, resp_headers = ParseResponseObject(row['response_object'])
132                 if type(row['receiver_data']) == bytes:
133                     received_data = row['receiver_data']
134                 elif type(row['receiver_data']) == str:
135                     received_data = row['receiver_data'].encode()
136                 else:
137                     log.error('Unknown type of "receiver_data": {}'.format(type(row['receiver_data'])))
138                     continue
139 
140                 item = CfurlCacheItem(row['time_stamp'], row['request_key'], http_req_method, req_headers,
141                                       http_status, resp_headers, row['isDataOnFS'], received_data,
142                                       username, app_bundle_id, cfurl_cache_db_path)
143                 cfurl_cache_artifacts.append(item)
```

Figure 12: The ParseCFURLEntry() Function

```
class CfurlCacheItem:
    def __init__(self, date, url, method, req_header, http_status, resp_header, isDataOnFS, received_data, username, app_bundle_id, source):
        self.date = date
        self.url = url
        self.method = method
        self.req_header = req_header
        self.http_status = http_status
        self.resp_header = resp_header
        self.isDataOnFS = isDataOnFS
        self.received_data = received_data
        self.username = username
        self.app_bundle_id = app_bundle_id
        self.source = source
```

Figure 13: Class that Holds Analysis Results

■ **ParseRequestObject() and ParseResponseObject()**

This function gets data from the request_object and response_object (Figure 14, Figure 15).

As noted above, the request_object array's 18th element holds the HTTP method, and the 19th holds the HTTP request header (lines 96–97 in Figure 14). We also exclude the __hhaa__ field (line 100 in Figure 14). Similarly, the response_object array's 3rd element is the HTTP status, and the 4th is the HTTP response header (lines 106–107 in Figure 15).

2.3.4 Saving the Analysis Results

■ **PrintAll()**

This function saves the analysis results in the format specified on the command line (Figure 16). cfurl_cache_info defines the column names and types used when storing the analysis results (lines 162–164). The analysis data items are collected into a list object in the same order as the items in the cfurl_cache_info definition, and in the final step, the WriteList() function writes the data to a file (lines 168–171).

2.4 Example of the Plugin in Action

FIgure 17 shows the analysis results from the plugin discussed here (columns to the right of Received_Data have been trimmed from the screenshot). I hope you'll agree that it is easy to examine the data once the information is organized like this.

```
94  def ParseRequestObject(object_data):
95      object_array = plistlib.loads(object_data)['Array']
96      http_req_method = object_array[18]
97      header_list = object_array[19]
98      req_headers = []
99      for header, value in header_list.items():
100         if header != '__hhaa__':
101             req_headers.append("{}: {}".format(header, value))
102     return http_req_method, "\r\n".join(req_headers)
```

Figure 14: The ParseRequestObject() Function

```
104 def ParseResponseObject(object_data):
105     object_array = plistlib.loads(object_data)['Array']
106     http_status = object_array[3]
107     header_list = object_array[4]
108     resp_headers = []
109     for header, value in header_list.items():
110         if header != '__hhaa__':
111             resp_headers.append("{}: {}".format(header, value))
112     return http_status, "\r\n".join(resp_headers)
```

Figure 15: The ParseResponseObject() Function

```
161 def PrintAll(cfurl_cache_artifacts, output_params, source_path):
162     cfurl_cache_info = [('Date', DataType.TEXT), ('URL', DataType.TEXT), ('Method', DataType.TEXT), ('Request_Header', DataType.TEXT),
163                         ('HTTP_Status', DataType.TEXT), ('Response_Header', DataType.TEXT), ('isDataOnFS', DataType.INTEGER), ('Received_Data', DataType.BLOB),
164                         ('User', DataType.TEXT), ('App_Bundle_ID', DataType.TEXT), ('Source', DataType.TEXT)]
165
166     data_list = []
167     log.info(f"{len(cfurl_cache_artifacts)} CFURL cache artifact(s) found")
168     for item in cfurl_cache_artifacts:
169         data_list.append([item.date, item.url, item.method, item.req_header, item.http_status, item.resp_header, item.isDataOnFS, item.received_data, item.username, item.
            app_bundle_id, item.source])
170
171     WriteList("CFURL cache", "CFURL_Cache", data_list, cfurl_cache_info, output_params, source_path)
```

Figure 16: The PrintAll() Function

	Date	URL	Method	Request_Header	HTTP_Status	Response_Header	IsDataOnFS	Received_Data
	フィルター	フィルター	フィル...	フィルター	フィルター	フィルター	フィルター	フィルター
1	2020-11-09 01:58:45	https://stackoverflow.com/	GET	Accept: */*...	200	Content-Encoding: gzip...	1	2B1680C0-DAE0-4EA0-9EC0-C4FC7F86A8C0
2	2020-11-09 02:10:43	https://www.example.com/	GET	Accept: */*...	200	Content-Type: text/html; charset=UTF-8...	0	<!doctype html>...
3	2021-02-02 06:58:03	https://raw.githubusercontent.com/its-a-feature/Orchard/master/Orchard.js	GET	Accept: */*...	200	Content-Encoding: gzip...	1	A391D5EC-9FCF-4993-A0AF-EEF2C871EF6A

Figure 17: Analysis Results from the Plugin

2.5 Conclusion

This two-part series has walked through the creation of a mac_apt plugin. While I have provided a broad understanding of plugin structure and process flow, I certainly have not covered all of the APIs mac_apt provides. Looking at other plugins also would be a great way to further your understanding.

mac_apt is a powerful forensic analysis tool, but the best way to get support for more artifacts is to write your own plugins. As explained in the previous installment, many artifacts are in plist or SQLite format, so it's very easy to look through the data, and the most important advantage is that you can analyze the data you require for your purposes in the format of your choice.

Finally, some readers may be more interested in how to go about reading and making sense of the mac_apt analysis results than in creating plugins. I would recommend the presentation slides[*8] and analysis data[*9] from the macOS hands-on forensics workshop given at the Japan Security Analyst Conference 2022 (JSAC2022) as a useful reference in this case. The workshop used mac_apt analysis results to create a forensic timeline of malware incursion. A video of the workshop is also available[*10].

Minoru Kobayashi
Forensic Investigator, Office of Emergency Response and Clearinghouse for Security Information, Advanced Security Division, IIJ
Mr. Kobayashi is a member of IIJ-SECT, mainly dealing with digital forensics. He works to improve incident response capabilities and in-house technical capabilities. He gives lectures and training sessions at security events both in Japan and abroad, including Black Hat, FIRST TC, JSAC, and Security Camp events.

*8 Workshop slides (https://jsac.jpcert.or.jp/archive/2022/pdf/JSAC2022_workshop_macOS-forensic_en.pdf).
*9 Analysis data (https://jsac.jpcert.or.jp/archive/2022/data/JSAC2022_macos_forensic_workshop_without_malware.7z).
*10 [JSAC2022] Workshop: An Introduction to macOS Forensics with Open Source Software (https://www.youtube.com/watch?v=Mor9EpInrXM).