Creating a mac_apt Plugin (Part 1)

2.1 What is mac_apt?

Digital forensics is so well provided for on Windows that free and open source tools alone are sufficient for analyzing most artifacts. Yet in the case of macOS—which, like Windows, is widely used as a desktop OS—relatively few commercial products, not to mention free and open source tools, are available.

This probably reflects the relative OS market shares and needs within the digital forensics market. But the last few years have seen the release of open source forensic analysis tools for macOS that implement just enough features to be practically useful. In my case, I have been following a tool called mac_apt[*1] closely.

This tool was developed as a macOS forensic analysis framework and can analyze a range of artifacts using over 40 plugins. It also implements its own APFS and HFS+ file system parsers, allowing direct analysis without the disk image mounted. So to compare it with tools designed to perform analysis on disk images mounted in the OS, mac_apt obviates the need to install file system drivers, and it allows the analyst to perform analysis regardless of what OS they are running on the analysis machine. And in addition to de facto standard disk image formats like RAW and E01, it also supports relatively niche formats like AFF4 and SPARSEIMAGE. These are used by commercial forensic tools as disk image formats.

Conversion tools can be used to convert disk images to any number of formats, so analysis tools do not necessarily have to support a whole bunch of formats. But because the disk images of computers these days often exceed several hundred GB in size, converting disk images takes a lot of time and disk space. So the ability to analyze disk images without converting them is a plus for the analyst.

Many plugins are implemented for mac_apt, allowing analysis of many key artifacts. But mac_apt is developed almost solely by its creator, Yogesh Khatri, who cannot be expected to provide support for all artifacts.

If an artifact is unsupported, it would perhaps be common to submit an Issue on the mac_apt development repository and wait for someone to volunteer to implement a plugin. But if you have some understanding of the artifact's data structure, you might also consider implementing the plugin yourself, because as mentioned earlier, mac_apt was developed as a forensic analysis framework.

This has been a somewhat lengthy preamble, but I will now go over the basics of creating plugins for the mac_apt forensic analysis framework for macOS. There is no official documentation on creating plugins, so what follows is based on the source code of plugins already implemented and what I have learned from creating plugins. While mac_apt is written in Python, I will not be explaining Python terminology and the like here.

2.2 Important File Formats for macOS Forensics

Before we get into creating plugins, let's look at file formats that are often parsed in macOS forensics. macOS and its applications use property lists (plists) and SQLite to store settings and history data. So naturally, artifact files often come in one of these formats (since forensic analysis often involves analysis of settings and history).

plist files are mainly used to store simple data like OS and application settings and history. They play a role like that of the Windows registry, but as they are created for each application, they are found in various places on the file system. Early plist files used an XML format, but a binary format is now the default. On the command line, plutil can

*1 macOS (& ios) Artifact Parsing Tool (https://github.com/ydkhatri/mac_apt).

© Internet Initiative Japan Inc.

be used to examine the contents of a plist file. Figure 1 shows an example of using plutil to display com.apple.dock.plist, which stores the settings for applications on the macOS Dock.

SQLite, like plist, is used to record settings and history, but it is also used to store slightly larger pieces of data such as blobs of sent and received data. It is also used in a range of applications including Chrome, and recently even in Windows some artifacts are saved in SQLite format. DB Browser for SQLite[*2] is a convenient way to view the data. Figure 2 shows an example of using DB Browser to read com.apple.LaunchServices.QuarantineEventsV2, which stores information related to the quarantining of files downloaded via a Web browser.

Figure 1: plist Example (com.apple.dock.plist)

テーブル: LSQuarantineEvent

カラムをフィルター

	LSQuarantineEventIdentifier	LSQuarantineTimeStamp	LSQuarantineAgentBundleIdentifier	LSQuarantineAgentName	LSQuarantineDataURLString	LSQu
	フィルター	>=663822201.0	フィルター	フィルター	フィルター	フィル
1	04C5C1E6-9DFA-460E-8D6C-96E9ED7FD561	663822201.0	org.mozilla.firefox	Firefox	https://2.na.dl.wireshark.org/osx/...	NULL
2	85681920-83B9-43D3-BA79-8E527BFE71B8	663869536.0	org.mozilla.firefox	Firefox	https://objects.githubusercontent.com/github-production-...	NULL
3	5A428EC4-53C5-4381-97E4-0BFDA8C4D011	664048278.955557	NULL	Homebrew Cask	https://github.com/rizinorg/cutter/releases/download/...	NULL
4	A71D48DE-C77F-4064-A748-33420CFC2642	664293755.627176	NULL	Homebrew Cask	https://downloads.sourceforge.net/grandperspectiv/...	NULL
5	440FC842-0E4F-4F97-9969-F428445E48E3	664870260.643495	NULL	Homebrew Cask	https://download.bell-sw.com/java/8u322%2B6/bellsoft-...	NULL
6	3D2AA369-7AAC-4F67-B2BF-6AF80FB3FC1B	665453437.78583	NULL	Homebrew Cask	https://downloads.sourceforge.net/grandperspectiv/...	NULL
7	7C673FC0-ED4C-44B8-A3F3-DD337448C259	665539858.0	com.google.Chrome	Chrome	https://optimizationguide-pa.googleapis.com/downloads?...	NULL
8	0A5AA187-241B-4F1C-9F89-49D0CBF69D3A	665970717.646598	NULL	Homebrew Cask	https://downloads.sourceforge.net/grandperspectiv/...	NULL

Figure 2: SQLite Example (com.apple.LaunchServices.QuarantineEventsV2)

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

© Internet Initiative Japan Inc.

2.3 mac_apt Plugin Structure

2.3.1 Demo Plugin

A demo plugin is provided in the mac_apt plugins folder in a file called _demo_plugin.py. This plugin reads the file "/System/Library/CoreServices/SystemVersion.plist", gets the value in ProductVersion, displays it on screen, and saves the analysis results to a file.

It only performs a simple analysis, but it is just right for understanding how plugins are structured, so let's use it as an example to see how plugins work in general.

2.3.2 Properties

Plugin properties are set near the beginning of the plugin (immediately after module imports) (Figure 3). Table 1 explains each of these properties.

Plugin authors can basically set these as they like, but __Plugin_Name needs to be unique as it is used to identify the plugin. __Plugin_Modes specifies what OS types (MACOS or IOS) the plugin supports. Note that the keyword "ARTIFACTONLY" can also appear in this property if you want to support the analysis of exported artifact files.

```
22  __Plugin_Name = "DEMOPLUGIN1" # Cannot have spaces, and must be all caps!
23  __Plugin_Friendly_Name = "Demo Plugin 1"
24  __Plugin_Version = "1.0"
25  __Plugin_Description = "Demonstrates logging, reading plist and writing out information"
26  __Plugin_Author = "Yogesh Khatri"
27  __Plugin_Author_Email = "yogesh@swiftforensics.com"
28
29  __Plugin_Modes = "MACOS,ARTIFACTONLY" # Valid values are 'MACOS', 'IOS, 'ARTIFACTONLY'
30  __Plugin_ArtifactOnly_Usage = 'Provide SystemVersion.plist to read macOS version'
```

Figure 3: Plugin Property Settings

Property name	Meaning	Example	Notes
__Plugin_Name	Plugin name	DEMOPLUGIN1	Must be all caps, cannot include spaces
__Plugin_Friendly_Name	Friendly name of plugin	Demo Plugin 1	Not used within the program
__Plugin_Version	Version	1.0	Not used within the program
__Plugin_Description	Plugin description	Arbitrary string	
__Plugin_Author	Author	John Smith	Not used within the program
__Plugin_Author_Email	Author's email	author@example.com	Not used within the program
__Plugin_Modes	OSs supported	MACOS,IOS,ARTIFACTONLY	
__Plugin_ArtifactOnly_Usage	Usage info for mac_apt_artifact_only.py	Arbitrary string	

Table 1: Meaning of Plugin Properties

© Internet Initiative Japan Inc.

2.3.3 Entry Points

All plugins first call the Plugin_Start(), Plugin_Start_Standalone(), or Plugin_Start_Ios() function. Table 2 lists the plugin entry points for different mac_apt commands.

The demo plugin implements two entry points, Plugin_Start() and Plugin_Start_Standalone(). And this is consistent with the content of the __Plugin_Modes property (Plugin_Start_Ios() contains only a pass instruction, and IOS does not appear in __Plugin_Modes). Next, let's look at what happens at each entry point.

■ Plugin_Start()

Plugin_Start() (Figure 4) takes a mac_info object as an argument. This object contains basic macOS information (OS version, user list, etc.) obtained from the disk image to be analyzed along with basic methods for accessing the files on the disk image.

The demo plugin displays the name of the OS on which mac_apt is running and the macOS version for which the analysis is being performed (lines 40–41). The function then sets the artifact file path, pulls the version number

mac_apt command	Plugin entry point
mac_apt.py mac_apt_mounted_sys_data.py	Plugin_Start()
mac_apt_artifact_only.py	Plugin_Start_Standalone()
ios_apt.py	Plugin_Start_Ios()

Table 2: mac_apt Commands and Entry Point Called

```
36  def Plugin_Start(mac_info):
37      '''Main Entry point function for plugin'''
38
39      # Lets print the macOS name and version that the framework has already retrieved. (Utilizing MacInfo)
40      log.info("Current OS is: " + os.name)
41      log.info("Mac version is : {}".format(mac_info.os_version))
42
43      # Now lets try to get it ourselves manually.
44      file_path = '/System/Library/CoreServices/SystemVersion.plist'
45      version = Process_File(mac_info, file_path)
46      log.info("Mac version retrieved = {}".format(version))
47
48      # Lets export our file into the Export folder, as most plugins should.
49      mac_info.ExportFile(file_path, __Plugin_Name)
50
51      # Let's write it out now
52      WriteMe(version, mac_info.output_params, file_path)
```

Figure 4: The Plugin_Start() Function

© Internet Initiative Japan Inc.

out of the artifact file using the Process_File() function, and displays this on screen (lines 44–46). It then exports the artifact file from the disk image and saves it in a folder with the same name as the plugin (line 49). Finally, it uses the WriteMe() function to save the analysis results (line 52).

So at the entry point, once the artifact file path has been set, the main tasks are to call a function that performs analysis and a function that saves the analysis results (see below for details of Process_File(), WriteMe()). Other plugins used to perform actual analysis also follow this same procedure.

The path of the artifact file analyzed by the demo plugin is a fixed string, but depending on the artifact, the file path could be undetermined. For example, if the artifact file is located within the user home directory tree, the file path will contain a user name and is thus not a fixed string. In such cases, you need to use the methods provided by mac_apt to get a list of directories and files on the disk image and dynamically build artifact file paths.

Plugin_Start_Standalone()

Plugin_Start_Standalone() (Figure 5) takes as its first argument a list object of artifact files specified on the mac_apt_artifact_only.py command line, so this can be iterated over to process the artifact files one after the other (line 96).

However, if the artifact is made up of multiple files, or if the settings result in the artifact file path(s) being

```
93  def Plugin_Start_Standalone(input_files_list, output_params):
94      '''Main entry point function when used on single artifacts (mac_apt_singleplugin), not on a full disk image'''
95      log.info("Module Started as standalone")
96      for input_path in input_files_list:
97          log.debug("Input file passed was: " + input_path)
98          ## Process the input file here ##
99          if input_path.endswith('SystemVersion.plist'):
100             success, plist, error = CommonFunctions.ReadPlist(input_path)
101             if success:
102                 os_version = plist.get('ProductVersion', None)
103                 if os_version == None:
104                     log.error('Could not find ProductVersion in plist!')
105                 else:
106                     WriteMe(os_version, output_params, input_path)
107             else:
108                 log.error('Input file "{}" is not a valid plist. Error opening file was: {}'.format(input_path, error))
```

Figure 5: The Plugin_Start_Standalone() Function

© Internet Initiative Japan Inc.

undetermined, you will again need to dynamically build the artifact file paths. Since the artifact files are on the file system of the OS running mac_apt, you can use Python's standard os module to get the file list and so on.

In the demo plugin, if the file path ends with "SystemVersion.plist", the file is parsed by the ReadPlist() method in the CommonFunctions module provided by mac_apt (line 100). If the file is successfully parsed, then after obtaining the OS version, the function saves the analysis results using the WriteMe() function (lines 101–106). As you can see, Process_File() is not called here, but the procedure is mostly the same as in Plugin_Start().

Note that the result of parsing a plist file using the ReadPlist() method (second return value) is a dictionary object.

2.3.4 The Demo Plugin's Other Functions

■ Process_File()

A function that parses SystemVersion.plist (Figure 6). Parses the artifact file passed in as the second argument using the mac_info object's ReadPlist() method (line 60). If successful, calls the GetMacOsVersion() function, described below, to get the version number and returns it (lines 61–65).

The mac_info object's ReadPlist() method, like that in the CommonFunction module, returns the result of parsing the plist as a dictionary object.

■ GetMacOsVersion()

Gets the OS version from parsed plist data and returns it (Figure 7).

```
55  def Process_File(mac_info, file_path):
56      version = ''
57      log.debug("Inside Process_File")
58      try:
59          log.info("Trying to get version from {}".format(file_path))
60          success, plist, error = mac_info.ReadPlist(file_path)
61          if success:
62              version = GetMacOsVersion(plist)
63      except Exception:
64          log.exception(error)
65      return version
```

Figure 6: The Process_File() Function

```
67  def GetMacOsVersion(plist):
68      ''' Gets macOS version number from plist, input here is the plist itself.'''
69      try:
70          os_version = plist['ProductVersion']
71      except Exception:
72          log.error("Error fetching ProductVersion from plist. Is it a valid xml plist?")
73      return os_version
```

Figure 7: The GetMacOsVersion() Function

© Internet Initiative Japan Inc.

WriteMe()

A function that saves the analysis results to a file (Figure 8). col_info defines the columns used when writing the analysis results (line 77). The definition is a list of tuple objects. The first element of each tuple is the column name and the second is the column's type. Common type values are "DataType.TEXT" and "DataType.INTEGER". In the demo plugin, the first column is named "Version info" and contains text, the second is named "Major" and contains integers.

The data variable holds the data (list object) to be saved (line 79). The length of this list must match the length of the columns definition.

The DataWriter object is used to save the analysis results in the location and file format specified on the mac_apt command line (line 82). The first argument holds information such as the save folder, the second holds the table name (when saving in SQLite format), the third is the columns definition, and the fourth is the artifact file path. But as the fourth argument is not used, you can simply pass in an empty string. The DataWriter object's WriteRow() method is used to actually write the data.

As the comment on line 90 indicates, however, the above process can also be accomplished in a single line using the WriteList () function. A look through other plugins reveals that most of them use the WriteList() function. The first argument is a string giving some details about

```
76  def WriteMe(version, output_params, file_path):
77      col_info = [ ('Version info', DataType.TEXT),('Major', DataType.INTEGER) ] # Define your columns
78      major_ver = int(version.split('.')[0])
79      data = [version, major_ver] # Data as a list (or dictionary)
80
81      ## The following demonstrates use of the writer class.
82      writer = DataWriter(output_params, 'macOS Info', col_info, file_path)
83      try:
84          writer.WriteRow(data)
85      except:
86          log.exception('WriteMe() exception')
87      finally:
88          writer.FinishWrites()
89
90      # Alternately, you could do it in one line as shown below:
91      WriteList('MacOS version info', 'macOS Info', [data], col_info, output_params, file_path)
```

Figure 8: The WriteMe() Function

© Internet Initiative Japan Inc.

the data, but this is only written to logs and is not saved in the file. The sixth argument can simply be an empty string since, within the WriteList() function, it is used as the fourth argument of a DataWriter object.

2.3.5 Naming Rules for Functions etc.

The entry point function names are fixed, but the plugin author needs to decide on the names of the other functions that perform analysis and save the analysis results. As noted earlier, there is no documentation on how plugins should be written, but looking at existing plugins, most seem to use Pascal case function names, and variable names are in snake case. This is good to be aware if you want to be consistent with other plugins.

It also looks like analysis functions are often named "ProcessXxxx()" or "ParseXxxx()". And "PrintAll()" is used consistently as the name of the function that saves analysis results (although the demo plugin uses WriteMe()). This function has three arguments. The first is a list object holding the analysis results to be saved. The second is an object (mac_info.output_params) that holds settings such as the save destination and save file format. The third is ultimately passed to the WriteList() function as its sixth argument, and as such, it appears to be an empty string in most cases.

© Internet Initiative Japan Inc.

2.4 Finding Artifacts Not Supported by mac_apt

As I mentioned, the creator of mac_apt maintains it almost single-handedly, so there are unsupported artifacts. So when looking at other analysis tools or reading macOS security articles, you may notice artifacts that mac_apt does not support.

For example, when reading an article about how an attacker could, as a persistence method, replace the path of an application in the Dock with the path of a malicious program[*3], I realized that mac_apt does not provide analysis of "~/Library/Caches//Cache.db". To check whether mac_apt supports a given artifact, you can look through the list of plugins or search the mac_apt source code for the name of an artifact file.

When I checked Cache.db on an actual machine, I found that it stores not only HTTP but also HTTPS traffic (Figure

テーブル: cfurl_cache_response

	entry_ID	version	hash_value	storage_policy	request_key	time_stamp	partition
	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	1	0	-6082800930625395189	0	https://stackoverflow.com/	2020-11-09 01:58:45	NULL
2	2	0	1647889406	0	https://www.example.com/	2020-11-09 02:10:43	NULL
3	3	0	2145575174	0	https://raw.githubusercontent.com/Its-a-feature/Orchard/...	2021-02-02 06:58:03	NULL

Figure 9: The cfrul_cache_response Table

*3 Are You Docking Kidding Me? (https://posts.specterops.io/are-you-docking-kidding-me-9aa79c24bdc1).

© Internet Initiative Japan Inc.

9). And not only that, it also stores the HTTP request method, HTTP status, HTTP headers, and response body (Figures 10 and 11). Information like this can be very useful when doing forensics, so it would be well worth thinking about implementing a plugin for this.

In the next issue of the IIR, I will discuss the data stored in Cache.db in detail and the implementation of a plugin for analyzing this artifact file.

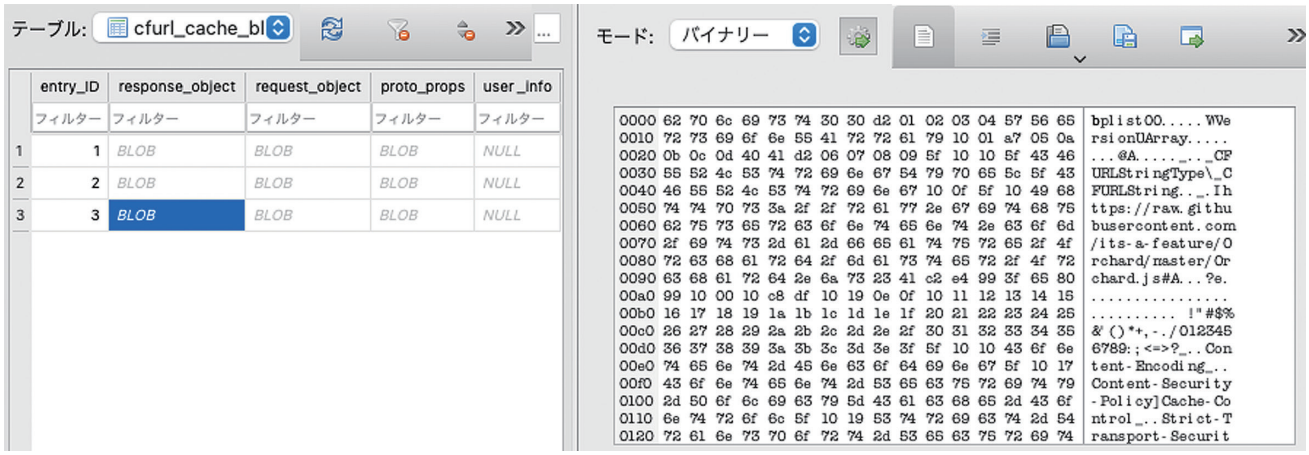

Figure 10: The cfurl_cache_receiver_data Table

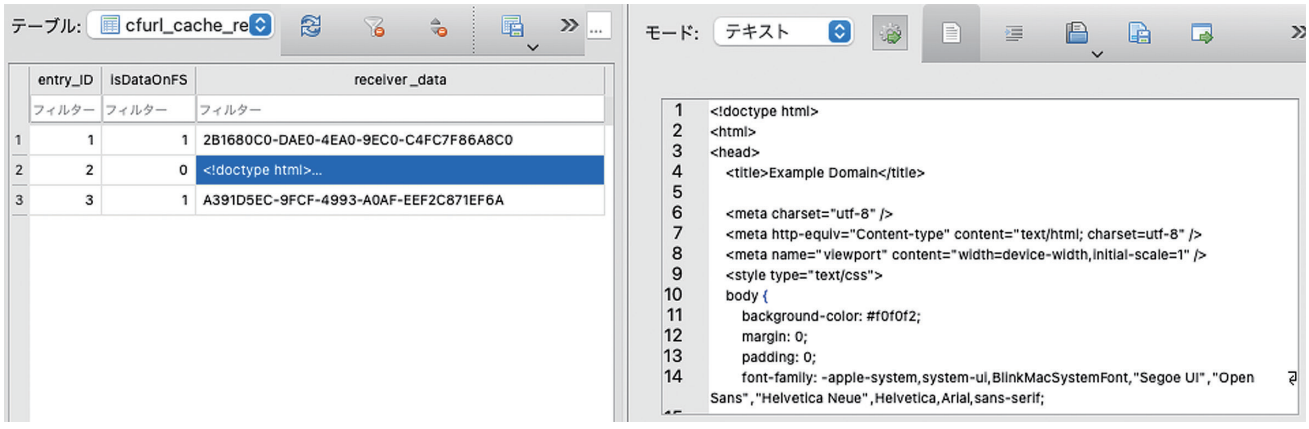

Figure 11: The cfurl_cache_blob_data Table

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.

© Internet Initiative Japan Inc.