International Journal of Digital Crime and Forensics
Volume 12 • Issue 1 • January-March 2020

Evaluation of Autopsy and Volatility for Cybercrime Investigation

A Forensic Lucid Case Study

Ahmed Almutairi, Concordia University, Quebec, Canada
Behzad Shoarian Satari, Concordia University, Quebec, Canada
Carlos Rivas, Concordia University, Quebec, Canada
Cristian Florin Stanciu, Concordia University, Quebec, Canada
Mozhdeh Yamani, Concordia University, Quebec, Canada
Zahra Zohoorsaadat, Concordia University, Quebec, Canada
Serguei A. Mokhov, Concordia University, Quebec, Canada

ABSTRACT

In this article, the authors successfully created two new plugins one for Autopsy Forensic Tool, and the other for Volatility Framework. Both plugins are useful for encoding digital evidences in Forensic Lucid which is the goal of this work. The first plugin was integrated in Autopsy to generate a report for the case of a Brute Force Authentication attack by looking for evidence in server logs based on a key search. On the other hand, the second plugin named ForensicLucidDeviceTree aims to find whether a device stack has been infected by a root-kit or not expression is implied by the previous statement. The results of both plugins are shown in Forensic Lucid Format and were successfully compiled using GIPC compiler.

KEYWORDS

Autopsy, Forensic Lucid, GIPC Compiler, Volatility

1. INTRODUCTION

1.1. Motivation

The motivation behind this project is to re-evaluate the open source forensic tools through their hands-on use, such as that of Sleuthkit (Carrier, n.d.), and more of its Autopsy (Carrier, n.d.), and other tools in a simulated investigation, reasoning, analysis, and reporting for sample cases. The use of tools is followed by adaptation and encoding of the case's knowledge base (output) extracted from forensic artifact analysis in Forensic Lucid. Thus, the tools should be evaluated how easy is to extract their outputs, reports, and translate into the format for Forensic Lucid. The sample data would come from the honeynet (Honeynet Project, 2015) and DFRWS (Palmer, 2001) projects/challenges.

1.2. Overview

In Section 2 we provide a detailed background of our research on Autopsy (Carrier, n.d.), Volatility, Forensic Lucid, and GIPSY, that has a Forensic Lucid compiler – GIPC. In Section 3 we detail our experiments, writing plug-ins for Autopsy and Volatility, and encoding sample data output into Forensic Lucid. We conclude in Section 5.

DOI: 10.4018/IJDCF.2020010104
This article, originally published under IGI Global's copyright on January 1, 2020 will proceed with publication as an Open Access article starting on January 27, 2021 in the gold Open Access journal, International Journal of Digital Crime and Forensics (converted to gold Open Access January 1, 2021), and will be distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

2. BACKGROUND

2.1. Autopsy

Autopsy is an Open Source application under the GNU General Public License Version 3. Autopsy forensics browser version 4.5.0 installs a Graphical digital forensics interface to the command line digital investigation tool Sleuth Kit and is built into the SANS investigative Forensic Toolkit workstation (SIFT Workstation) that is downloadable from https://www.autopsy.com or https://www.sleuthkit.org/autopsy/. Autopsy can be installed under Microsoft Windows 7/8.1/10, Linux Ubuntu and Mac OS X.

Autopsy tool allows to examine a hard drive or mobile device and recover evidence from it. The data sources which is supported by this tool are Disk images and VM file (e.g. Raw images (img, dd, 001, aa, raw, bin), Encase Images (E01), Virtual Machine images (vmdk, vhd formats), Local disk (local PC/server/laptop disk) and logical files (local files and directory to be added).

The ingest modules in Autopsy are responsible for the data analysis and subsequent data extraction for operations such as tagging and reporting. The tool has the following standard Ingest modules: File Type Identification, Recent Activity, and Hash database lookup, Embedded File Extractor, EXIF Parser, Email Parser, Virtual machine Extractor, and Photo Record Carver. Autopsy supports 3rd party ingest module such as Python plugins, Pre-fetch parser, Windows registry ingest modules, Virus total online checker, and Image fingerprint module.

Autopsy uses keyword search to analyze the disk image contents. The standard keyword list in autopsy includes IP addresses, phone numbers, email addresses and credit card numbers, and URLs. The group of keywords can be defined as an exact match, a substring match, or as a regular expression. The group of keywords will be created to look for a specific cyber offense such as a brute force authentication, spamming or click-thru fraud, found within an apache web server log files.

Tagging of data can be done on the files of the data source or on the search results such as keyword hits, email addresses, extracted content, interesting items, and accounts. Autopsy has two tagging categories: tagging by file (i.e. the file that includes the result) and tagging by result (e.g., keyword hits within each file). Once the items are tagged, they are added automatically into a user interface section called Tagged Results.

Autopsy offers the possibility to create a timeline for all events from the files on the disk. Furthermore, Autopsy has the capability of creating reports out of analyzed data and any tagged files. The investigator also can create a custom result by adding the tagged files showing the facts regarding certain cyber offenses. The reports in Autopsy can be generated in various formats such as HTML, Excel, CSV, Text, and tagged hashes, among others.

In this project, we have considered Autopsy version 4.5.0, However Autopsy has released version 4.6.1 on Feb. 2018. In order to gather data on some types of attacks, we used a honeypot image that is dumped from an Apache web server downloaded from apache_logs.tar.gz. There are four types of attack that can be examined on the mentioned data source:

1. Brute force Authentication: This type of attack is launched through the server by HTTP GET Requests and HTTP POST requests. It could be detected by examining the audit files.
2. IRC connection: IRC (Internet Relay Chat) is a protocol for real-time text messaging between internet-connected computers. It is mostly used for hiding the real IP address to launch denial of service attack. This type of attack could be detected by searching the audit log files for all entries to common IRC ports.
3. Spamming: A large number of users were spammers trying to send their emails through the server to hide their true location and make the tracking of the email's origin difficult.. This attack could be identified by examining the audit log files.

4. Click-Thru Fraud: This type of attack is not a normal web attack and is generated by banner advertisements and pay-per-click hyperlinks. The attack can be detected if the source IP address of a banner is repeated too many times within a period of time. It could be determined by scanning the audit log files.

2.2. Volatility

Volatility Framework is an open source framework implemented in Python under the GNU General Public License version 2, for Random Access Memory (RAM) analysis for 32bit and 64bit Windows, Linux, Mac and Android systems[1]. The performance of analysis technique is completely independent of the system being investigated so Volatility supports new operating systems and architectures as they are released. The most recent version (Volatility 2.6) was released on December 2016[2] and the older versions are also available on the Volatility foundation website.

For installation, Volatility runs on any platform that supports Python, in details Volatility has several installation options including (i) source code in zip/tar archive for Windows, Linux and OSX, (ii) Pyinstaller executable for Windows-only and (iii) standalone executables for Windows, Linux and MacOS[3].

The file formats that Volatility can analyze are raw dumps, crash dumps, hibernation files, VMware vmem, VMware saved state and suspended files (.vmss / .vmsn), VirtualBox core dumps, LiME (Linux Memory Extractor), expert witness (EWF), and direct physical memory over Firewire[4].

There are a large number of plugins in Volatility that are developed by Volatility foundation and by the community, these plugins (e.g., *malfind, cmdscan, apihooks, impscan*[5]) facilitate forensic analysis as well as development of forensic techniques and are able to detect malware that may have several layers of encoding obfuscations (e.g., *ZEUS*)[6]. A key aspect used in our project is the Volatility scriptable API that allows a developer to use Volatility to investigate kernel memory, to add new plugins, to perform virtual machine and to run a malware sandbox.

The pros of Volatility are the fast analysis memory dumps from large systems without unnecessary memory consumption. For example, Volatility is able to list kernel modules from an 80 GB system in just a few seconds. However, it takes several hours to do the same thing on a much smaller memory dump by other memory analysis frameworks. Volatility is supported by malware experts and forensic analysts from SANS institute and it is used as a one of the packages in the SIFT workstation.

2.3. Forensic Lucid and GIPSY

Lucid family are dataflow programming languages designed to model and verify different phenomena that could include other domains such as tensor fields, circuits, etc. (Edwards, Townsend, & Kim, 2017).

The research started by defining the background required for Implementation and verification of Forensic Lucid encoders for different popular server software as plug-ins or modules to provide functionality to the said servers to log their data directly in Forensic Lucid and/or write translation tools (scripts) to translate existing logs into Forensic Lucid (Mokhov, 2013; Mokhov, Assels, Paquet, & Debbabi, 2014).

The Forensic Lucid compiler is in General Intentional Programming System's GIPC framework. GIPSY also has a General Education Engine (GEE) runtime implemented using various distributed middleware technologies for scalable evaluation and comparison. It is extensible allowing adding Lucid and non-Lucid demand-driven evaluation of requests and their storage for the requests that were already processed under a certain context (Vassev & Paquet, 2008; Paquet, 2009; Paquet & Kropf, 2000; Ding, 2004; Tao, 2004; Wu, 2009; Han, 2010; Ji, 2011; Vassev, 2005; Tong, 2008; Pourteymour, 2008). The open source Forensic Lucid examples used for inspiration are found in https://github.com/smokhov/atsm/tree/master/examples/flucid.

3. METHODOLOGY

3.1. Motivation

We are motivated to provide some tools to forensic investigators in order to facilitate their work and be able to produce a compelling forensic analysis report. We decided to cover a wider area of digital evidence such as servers and service logs, and desktops' memory dumps/images; all these ones collected after the fact for dead forensics. We have written two plugins:

- A Java plugin as a report module for Autopsy that uses log files from an open proxy server (e.g., proxy-pot, honey-pot).
- A Python plugin in Volatility that uses memory images from Windows workstations infected with malware (e.g., rootkit).

In both cases, we are identifying the evidence for offenses by doing keyword search and usage of regular expressions.

Our plugins help to extract evidential information from the aforementioned data sources and to encode them in a Forensic Lucid Format with two goals in mind: (i) firstly, to find a possible sequence of events that proves the cyber-crime; and (ii) secondly, to formalize the evidence in an output format that can be used in a Forensic Lucid processor (e.g., GIPSY (Mokhov, 2013)) together with other encoded evidence.

3.2. Autopsy

Autopsy's Keyword Search ingester is a powerful tool to analyze from an entire disk image to a set of log files. The standard keyword lists in Autopsy include IP addresses, phone numbers, email addresses, credit card numbers, and URLs. The group of keyword entries can be defined as an exact match, substring match, or a regular expression.

In our simulated case, a keyword entry will be created to look for a specific cyberoffense such as a brute force authentication attack into Apache web server's log files. Following, we will take advantage of tagging of results to pinpoint suspicious patterns related to cyberattacks. Once some items have been tagged, we will benefit from our developed plugin to output a report in the Forensic Lucid format for further processing.

Before looking into how a set of steps along with our developed plugin can help to report interesting information in the Forensic Lucid format (see Section 3.2.3 at the page 10), however, in the next two section, we start by reviewing the development aspects of our plugin.

3.2.1. Development Environment

By the time of this writing, Autopsy is written in Java 1.8 and uses the Netbeans Platform 8.27 to support rich-client applications. This platform provides an extension mechanism called modules. A module can be considered as a plugin to externally enhance the target application. Off the shelf, Autopsy is comprised of the following modules and reports:

- Core modules:
  - Autopsy-Core.
  - Autopsy-CoreLibs.
- Ingester modules:
  - Emails parser.
  - Experimental.
  - ImageGallery.
  - Keyword Search.

  - RecentActivity.
- Reports:
  - Results - HTML.
  - Results - Excel.
  - Add Tagged Hashes.
  - Files - Text.
  - Google Earth/KML.
  - STIX.
  - TSK Body File.

Reports are aimed at describing references of tagged files having comments and notes provided by the investigator and/or by automated searches performed during the ingestion process. These references include, among others, recent documents, keyword hits, web history, cookies, downloads, installed programs, attached devices, hashset hits, and search queries.

According to the developer's guide[8], to be able to start any development, two requirements are needed: the Netbeans IDE (8.2 or above) and a copy of the Autopsy platform. The latter can be downloaded in binary form or its source code can be built. In any case, the actual Netbeans module project must refer to this platform locally.

To develop a Report module, the Developing Report Modules[9] section mentions that there are three ways to do it by implementing specific Java interfaces:

1. org.sleuthkit.autopsy.report.TableReportModule
2. org.sleuthkit.autopsy.report.FileReportModule
3. org.sleuthkit.autopsy.report.GeneralReportModule

However, the only interface that seems to be public for the version 4.5.0 is GeneralReportModule. This interface exposes only two methods:

1. org.sleuthkit.autopsy.report.GeneralReportModule.generateReport(String reportPath, ReportProgressPanel progressPanel)
2. org.sleuthkit.autopsy.report.GeneralReportModule.getConfigurationPanel()

This means that the plugin must query the platform for any available data and must control certain aspects of the UI which requires more coding compared to the TabularReport option.

For more information on building Netbeans platform modules, consult the website[10].

3.2.2. Plugin Development Process

To assess how easy it would be to extend this tool and develop an encoder for *Forensic Lucid*, we approached it in two phases:

Firstly, it was decided to build a simple proof of concept to understand the internal workings of the Netbeans platform. To that end, a report plugin based on the TableReport interface was coded. This interface was public until the version 3.0.10 of Autopsy. The plugin is able to dump the Keyword Search hits which is fed with regular expressions.

The plugin code, when compiled, a *.NBM* file is created which is a self-contained plugin component ready to be installed through the Netbean platform's standard mechanism.

Plugin installation. The process to install any Netbeans plugin is very straightforward:

1. Store the .NBM file at a known location.
2. On Autopsy, select the Tools > plugins menu.

3. On the Downloaded tab, click on Add plugins... button. Look for the .NBM file on the filesystem and choose it.
4. On the same tab, click on Install button.

To test this plugin, click on the Generate Report button on the toolbar. A dialog showing a list of reports will be displayed. The plugin inserts an entry called Custom Report - 6610

(in the final version, this entry is called Forensic Lucid Encoding), choose it and follow the steps of a short wizard. For a reference, see Figures 1a, 1b, and 1c.

Secondly, since the TableReport interface is not available for version 4.5.0 (at the time of this writing), a encoder module/plugin for outputting *Forensic Lucid* observations that is built on top of the Autopsy's reporting engine, was coded by implementing the org.sleuthkit.autopsy.report. GeneralReportModule Java interface. This gave a little more control over the generation process, but it required more coding. Following is a UML sequence diagram (Figure 2) that depicts the main components involved in this functionality. The flow of messages is as follows:

1. The Report engine (ReportGenerator) instantiates an object of ForensicLucidConfigPanel to allow the user to specify (see Figure 10b) a global prefix, a global suffix, and a summary suffix for the names of the observation variables. There are mainly two types of observation variables: a sequence that refers to the actual observation variables and many other sequences (as many as keyword lists exist) having the detailed keyword hits (see listing 1).
2. The Report engine (ReportGenerator) is responsible for instantiating our ForensicLucidEncoder according to the Netbeans platforms rules (this plugin is at the *services* layer).
3. As our component has a graphical user interface, it must control an object of ReportProgressPanel to show progress of this lengthy task.
4. At this type, the method writeEncoding encapsulates all the logic to generate the *keyword hits* as *Forensic Lucid observations*.
5. To prepend all variables in the output file, a prefix is asked from the GUI (ForensicLucidConfigPanel. getPrefix).
6. To append the same suffix to each observation variable in the output file, a string is obtained from ForensicLucidConfigPanel.getSuffix.

Figure 1a. Autopsy's proof of concept development screenshots

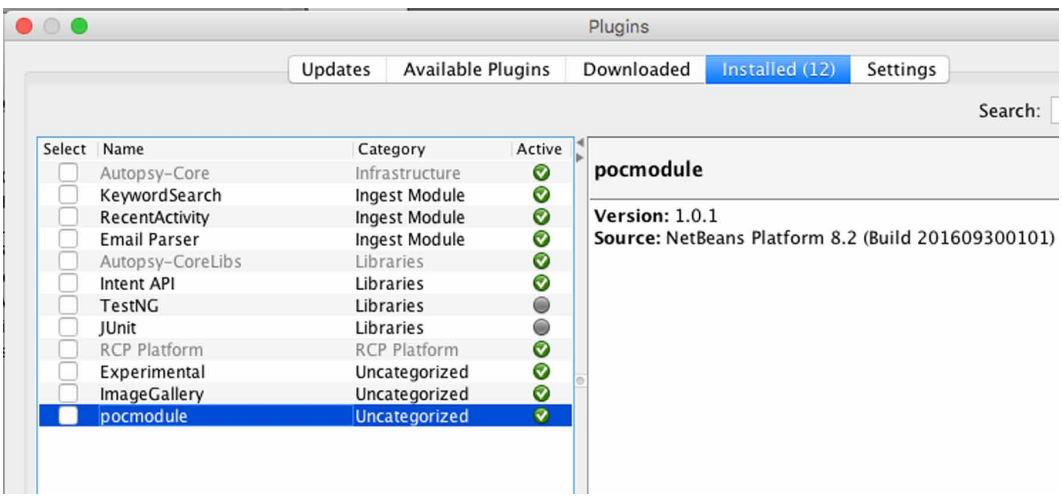


Figure 1b. Autopsy's proof of concept development screenshots

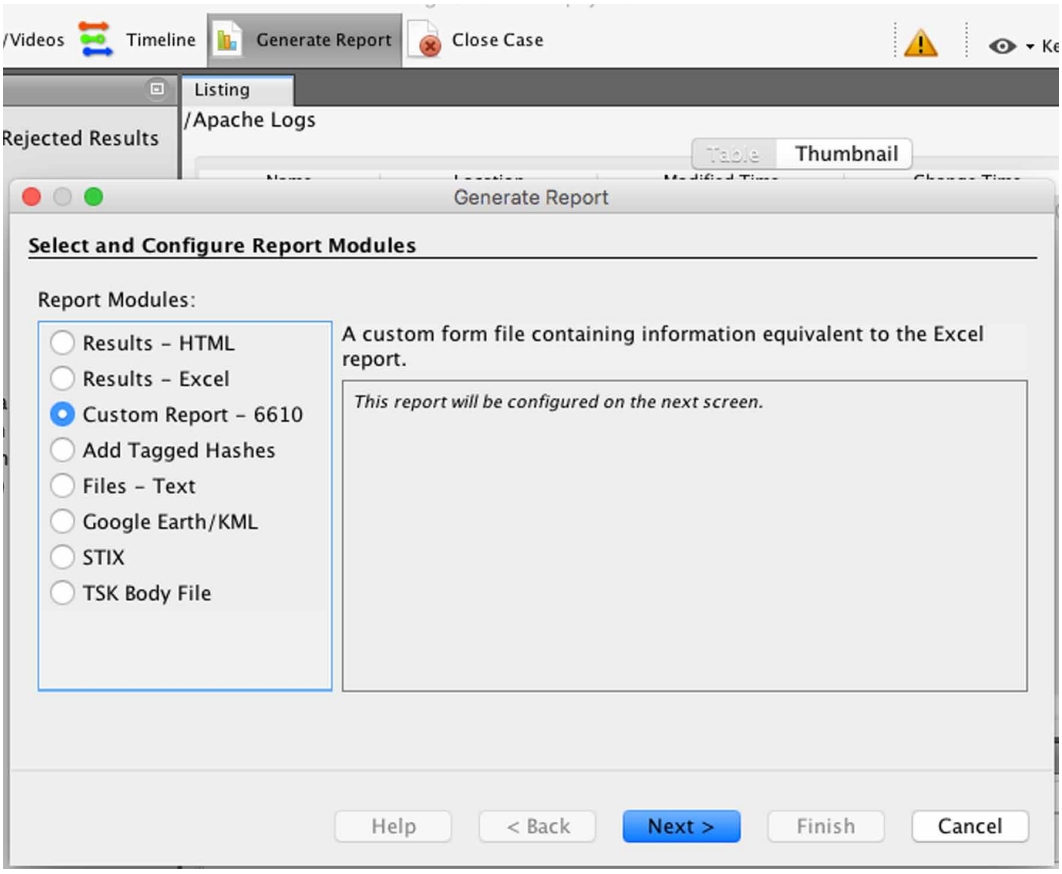


Figure 1c. Autopsy's proof of concept development screenshots

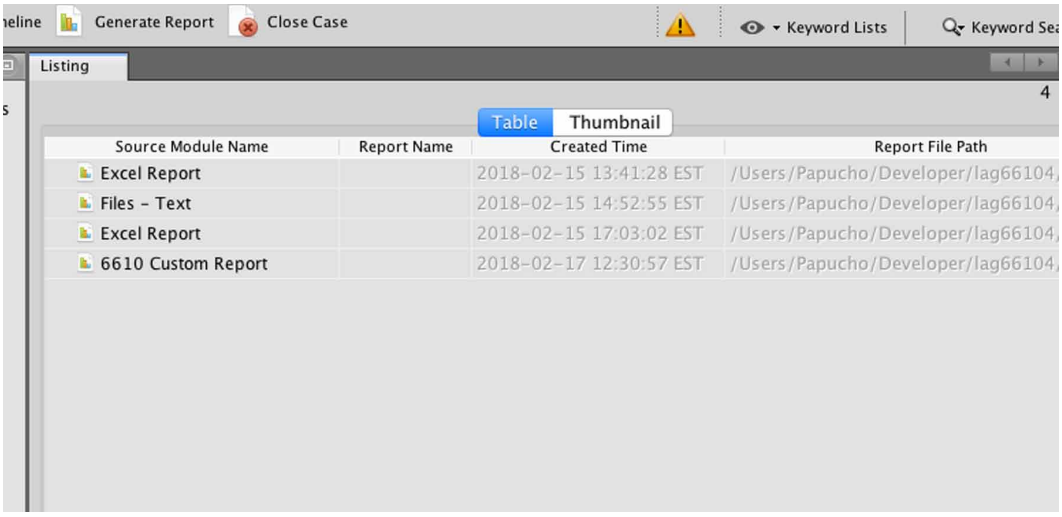
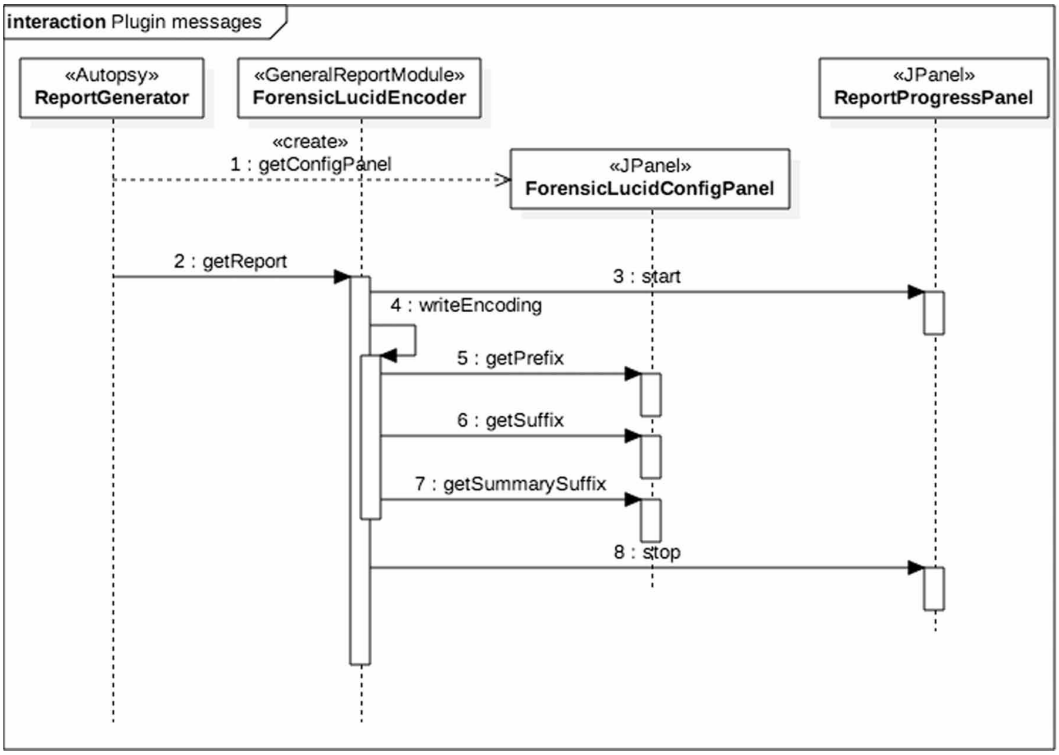


Figure 2. Autopsy's encoder messages



7. To append a suffix to the summary observation variable, a string is obtained from ForensicLucidConfigPanel.getSummarySuffix.
8. When the task is over, the GUI's progress status is updated via the ReportProgressPanel object.

As a last feature, this component is *i18ned*. To generate translated versions of it, use the /src/ca/concordia/c6610/report/Bundle.properties file as a template according to the standard Java i18n mechanism as implemented in the Netbeans platform[11].

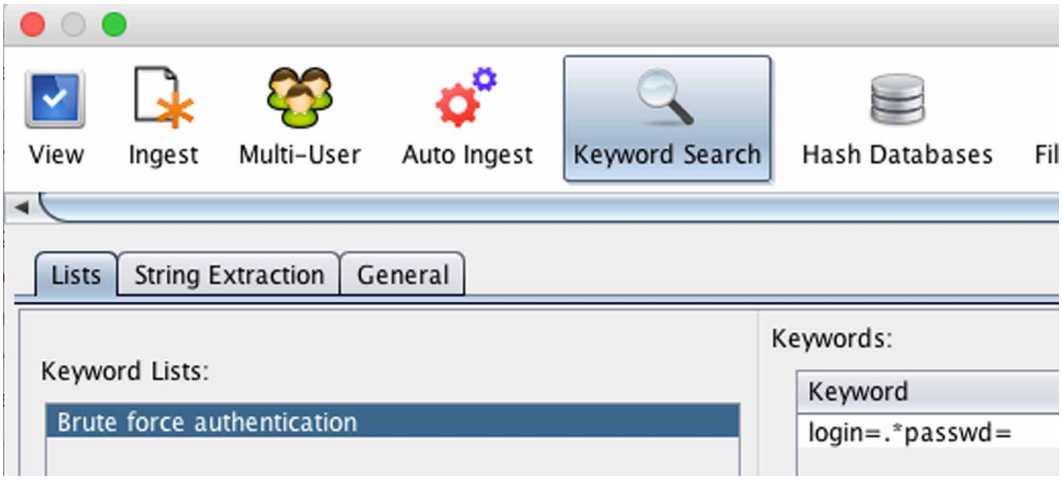
In the next section we will go over the steps on how to use this plugin to export specific observations produced by the Keyword search ingester module under a scenario of bruce force authentication attack.

3.2.3. Generation of the Forensic Lucid observations

Our simulated case helps to illustrate how an investigator could use the custom Forensic Lucid Encoding report plugin to ease the formalization of a brute force authentication attack observations. To that end, we have relied on the reasoning and data source found in honeynet.org[12]. As determined by this solution, Apache web server's logs can be analyzed for authentication attempts at *HTTP GET* requests, *HTTP POST* requests, and HTTP Basic Authentication headers.

Step 1: **Defining one or more regular expressions** When examining *HTTP GET* requests on Apache web servers, for instance, a common regular expression to search for is login=.*passwd=. Feeding the Keyword Search ingester module with this expression allows an investigator to identify text files whose contents matches it (any number of expressions can be indicated in a *keyword list*, though). Eventually, these results can be tagged at two levels (at a single entry and/or at the whole file).

Figure 3. Autopsy's keyword search setting



First of all, a keyword list entry must have been previously created. For example, a brute force authentication entry was added to specify the aforementioned regular expression (see Figure 3) using the Preferences > Keyword Search tab.

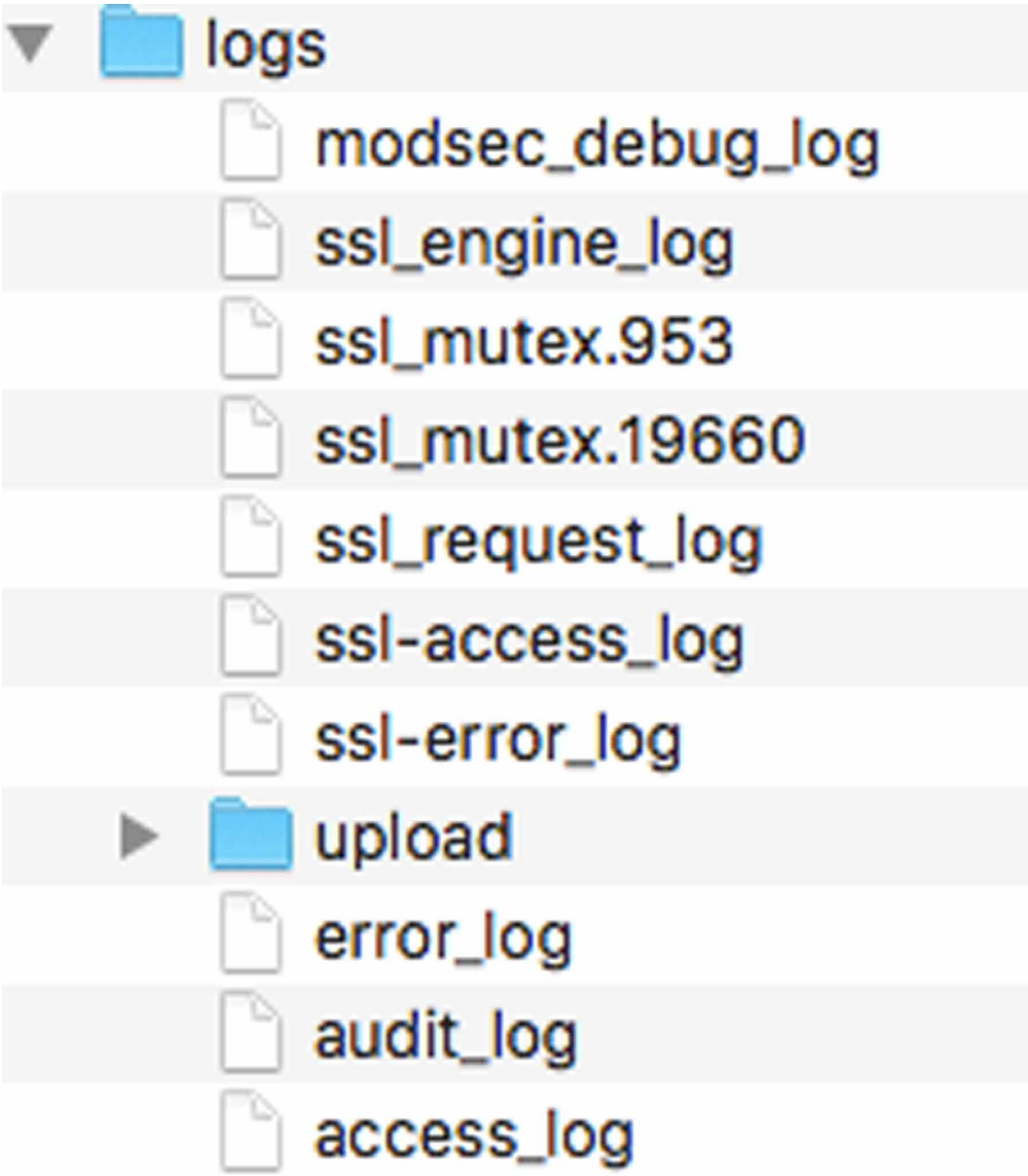
Step 2a: Set up the data source If the data source has not specified yet, before applying any ingester module, we must set up a data source. To do so, follow these instructions (see Figure 4):

1. Download the apache logs.tar.gz image12, ungzip it, and untar it (a log folder will be created eventually).
2. Add a data source (click on the Add Data Source button at the toolbar) to an existing case (otherwise, create a new case using the Case > New Case... menu items), select Logical Files and press the Next button, and on the next dialog, choose the uncompressed folder. Press Next button again.
3. Leave the Keyword Search box as well as the Brute force authentication box (*keyword list* section on the right) checked. Press Next and Finish buttons to start the ingestion process.
   (a) Wait for this process to end.

Step 2b: Run the Keyword Search ingest module For an existing data source (in an existing case) on which the Keyword Search ingester module has not run yet to scan all its contents, it is necessary to do so. To proceed, click on Tools > Run Ingest Modules and select a data source. In the Run Ingest Modules dialog (see Figure 4c), make sure to tick the Keyword Search box as well as the Brute force authentication box (keyword list section on the right), then click on the Finish button.

In any case, note that the ingestion process may take a long time to complete. Once finished, on the left-side panel (see Figure 5), on the Results > Keyword Hits > Brute force authentication node, the number of occurrences is shown along with the current regular expression as a sub-node. By expanding this last sub-node, all the matched substring is listed (see Figure 6). In this view, we can observe that there exists a certain pattern of how the login text is formed which might suggest a possible dictionary-based attack, so we should proceed to tag this result as a specific observation.

Figure 4a. Setting up a data source for use in the Autopsy's Keyword Search ingester



Step 3: Tag results of interest To tag an observation of interest, right-click on the target node on the left-side panel, then select Tag Result > Quick Tag > New Tag... context menu item as shown in Figure 7. Next, a dialog asking for a name for the tag is displayed (Figure 8a). Specify a name such as HTTP GET REQUEST e1.bjs.yahoo.com. This text is arbitrary, but meaningful to the investigator; it works as a mere reference of what is being observed (in this case, the type of HTTP request and the target server). We can validate if this tag was successfully created by going to Preferences... dialog, then click on the Tags icon (see Figure 8b). Repeat this tagging operation as needed. In this case, we have tagged another result using the same recently created tag as shown in Figure 9).

Figure 4b. Setting up a data source for use in the Autopsy's Keyword Search ingester

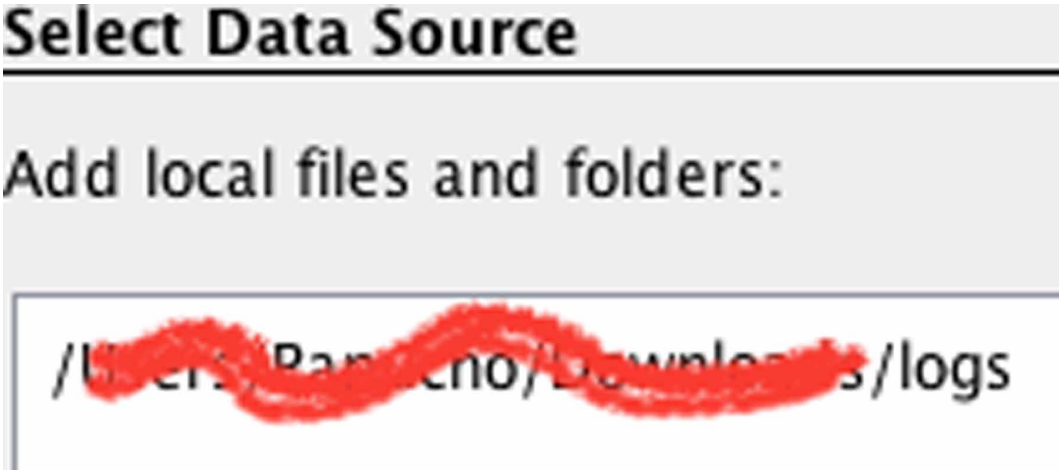
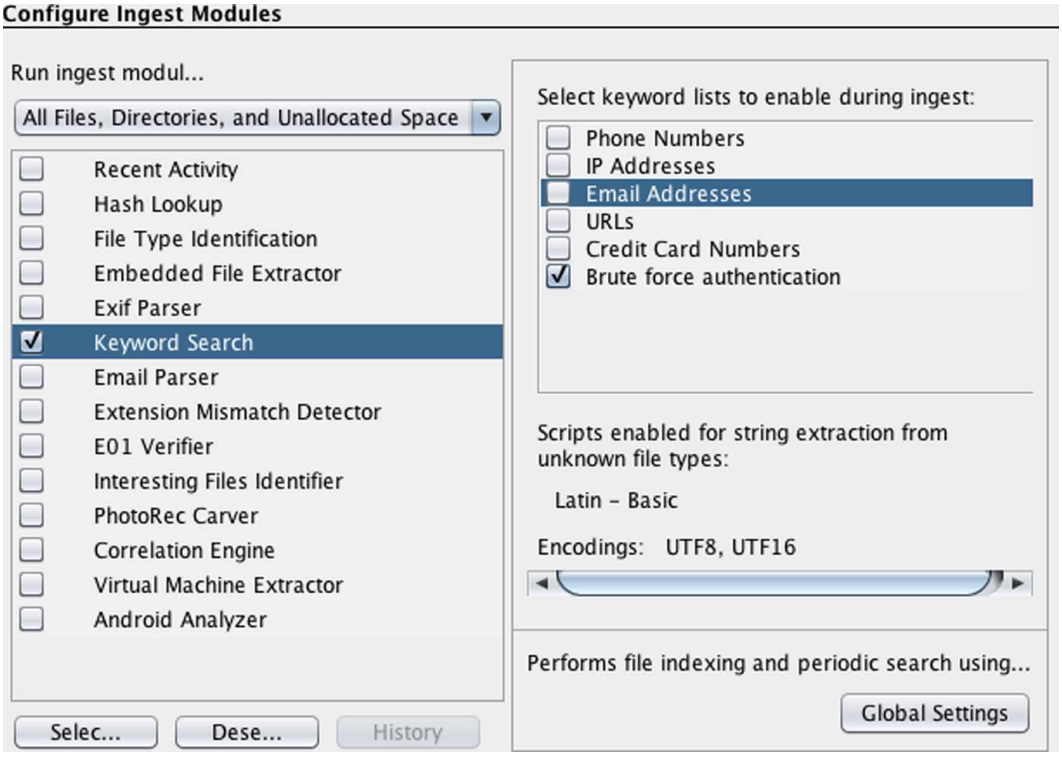


Figure 4c. Setting up a data source for use in the Autopsy's Keyword Search ingester



Step 4: Report the tagged observations. Given that all the prerequisites are met, a report in the Forensic Lucid format can be generated. In the Figure 10, a sequence of steps is depicted to show the report generation process from the user interface point of view:

Figure 4d. Setting up a data source for use in the Autopsy's Keyword Search ingester

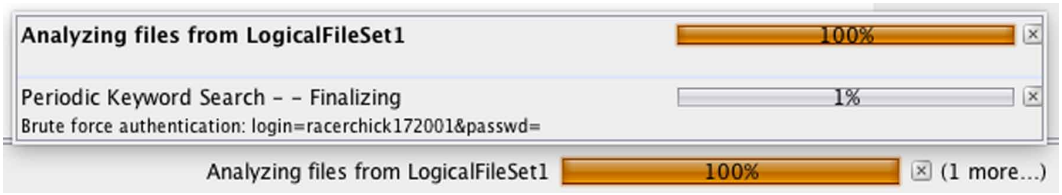


Figure 5. Keyword hits node

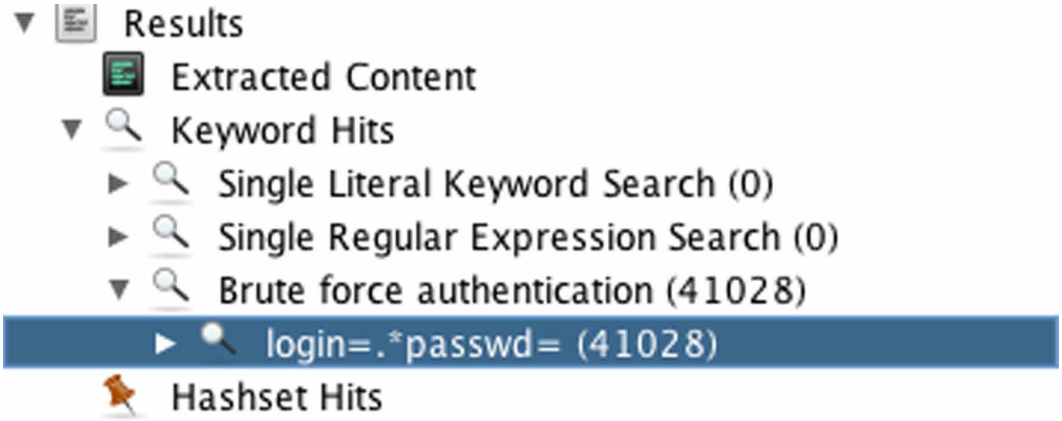
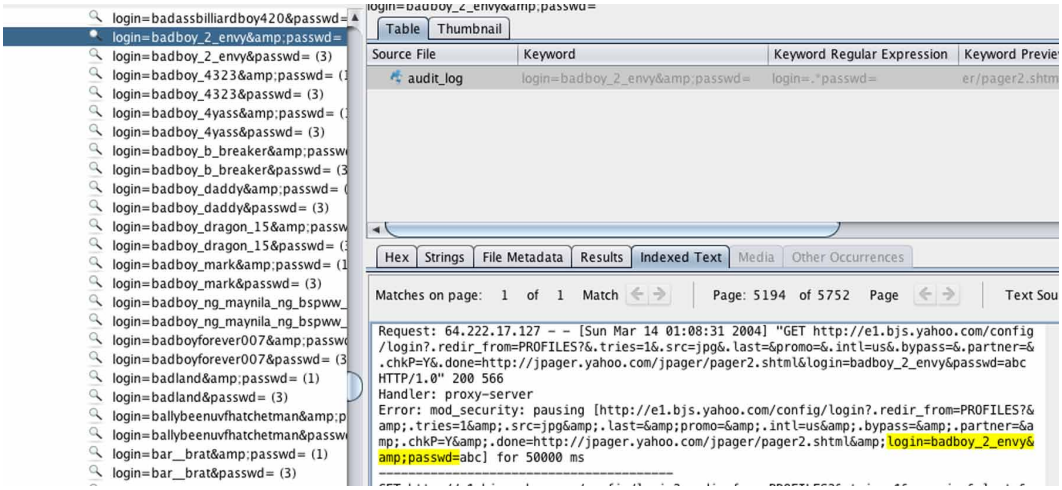


Figure 6. Keyword hits' search results



1. Once clicked on Generate Report button, the Forensic Lucid Encoding option must be chosen under the Report Modules section.
2. In the report, each observation (variable) is named after the concatenation of the keyword search entry. To customize this naming schema, on the left panel there are three parameters that can be set: The Global prefix for all variables text, if specified, allows to add a prefix to all observation

Figure 7. Tagging a keyword search result in autopsy

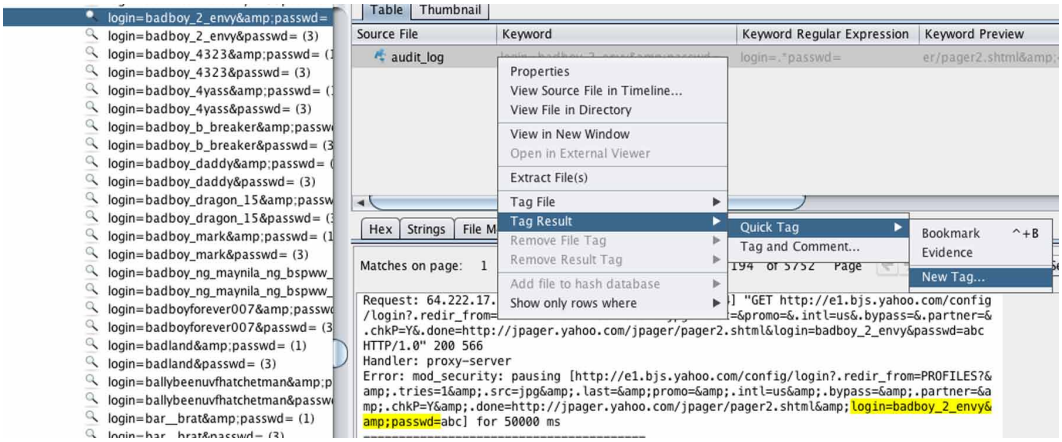
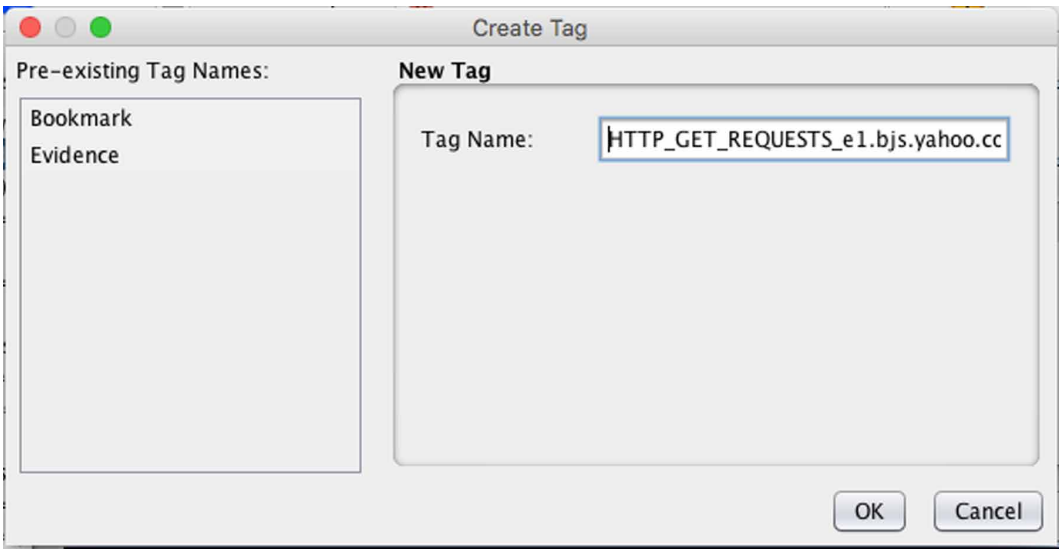


Figure 8a. Tag creation in autopsy



containers (variables). The Suffix for each keyword set variable text, if set, appends this text to each observation (sequence) corresponding to a keyword search entry. The Suffix for keyword summary variable text, if used, appends this text to the name of the container variable holding other observation sequences.

3. Next, a progress dialog is shown and displays the name of the Forensic Lucid observation report file as well as the status for each keyword search entry being processed.
4. Once all keyword search entries have been processed, the indicator is accordingly updated. At this moment, you can click on the Close button to dismiss the progress dialog (you can click on the Cancel button to stop processing at any time, however).
5. If the generation was successful, an entry is added under the Reports node (at the left panel of the main window). By clicking on this node, you will find an entry named as Keyword Hits as Forensic Lucid observations. This entry describes also the file's physical path.

Figure 8b. Tag creation in autopsy

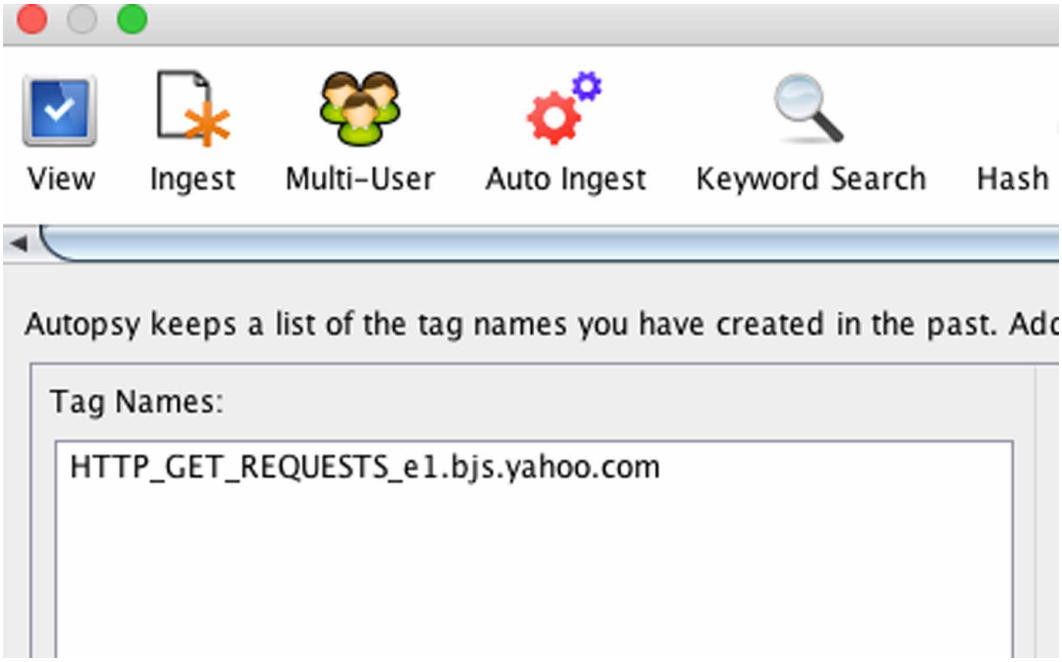
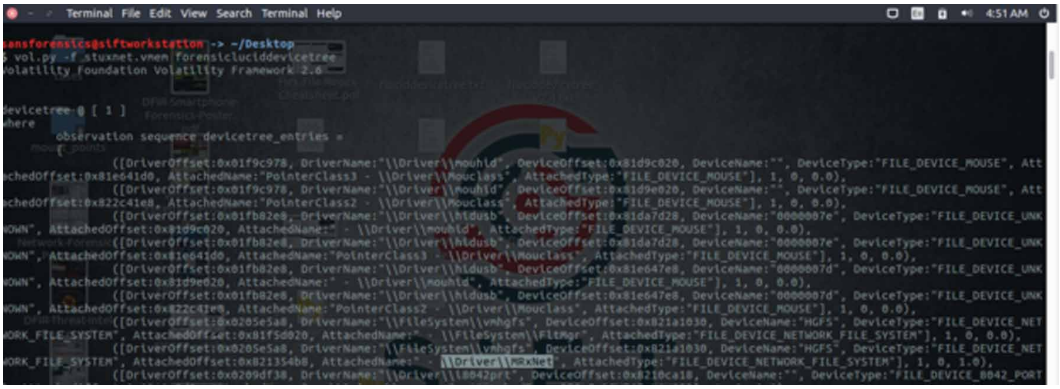


Figure 9. Reusing an existing tag in autopsy



In the listing 1, a sample report is shown which was obtained from some tagged observations as depicted in Figure 11.

```
} ;
end
```

As a result of the report generation process (page 12), a general observation sequence variable name is created by concatenating

```
<global prefix><keyword list entry title><suffix for summary
variable>
```

Resulting in observation sequence Autopsy_KeywordHits_Types = ...

Listing 1. Forensic Lucid observations from the Keyword Search hits

```
//************************* SUMMARY ******************************/
//Plugin version: 1.2.1
//Case Name: lag6610 -1
// Case Number: Case one
//Case Examiner: Carlos Rivas
//Num Of Images: 1
//*******************************************************************

Autopsy keywordsearches where
   //These types refer to this configured set of keywords: [Brute force authentication]observa
   tion sequence Autopsy KeywordHits Types =  -
   {
      Autopsy_BruteForceAuthentication Entries
   };

   //Observations for the "Brute force authentication" set (Keyword Hits artifact).observa
   tion sequence Autopsy BruteForceAuthentication Entries = -
   {
      (
         [
            Match: "login=badboy 2 envy&amp;passwd=",
            Preview: "er/pager2.shtml&amp;"login=badboy 2 envy&amp;passwd="abc] for 50000 ms--",SourceF
            ile: "/LogicalFileSet1/logs/audit log",      -
            Tags: "HTTP GET REQUESTS e1.bjs.yahoo.com"
         ],
         1.0,1.0
      ),(
         [
            Match: "login=badboy 2 envy&passwd=",
            Preview: "jpager/pager2.shtml&"login=badboy 2 envy&passwd="abc HTTP/1.0¥" 200 56",SourceF
            ile: "/LogicalFileSet1/logs/audit log",      -
            Tags: "HTTP GET REQUESTS e1.bjs.yahoo.com"
         ],
         1.0,1.0
      ),(
         [
            Match: "login=badboy 4323&amp;passwd=",
            Preview: "er/pager2.shtml&amp;"login=badboy 4323&amp;passwd="abc] for 50000 ms--",SourceF
            ile: "/LogicalFileSet1/logs/audit log",      -
            Tags: "HTTP GET REQUESTS e1.in.yahoo.com"
         ],
         1.0,1.0
      ),(
         [
            Match: "login=badboy 4323&passwd=",
            Preview: "jpager/pager2.shtml&"login=badboy 4323&passwd="abc HTTP/1.0¥" 200 56",SourceF
            ile: "/LogicalFileSet1/logs/audit log",      -
            Tags: "HTTP GET REQUESTS e1.in.yahoo.com"
         ],
         1.0,1.0
      )
```

This container holds an element for each keyword list entry configured in Autopsy. Each observation sequence variable name is created by concatenating

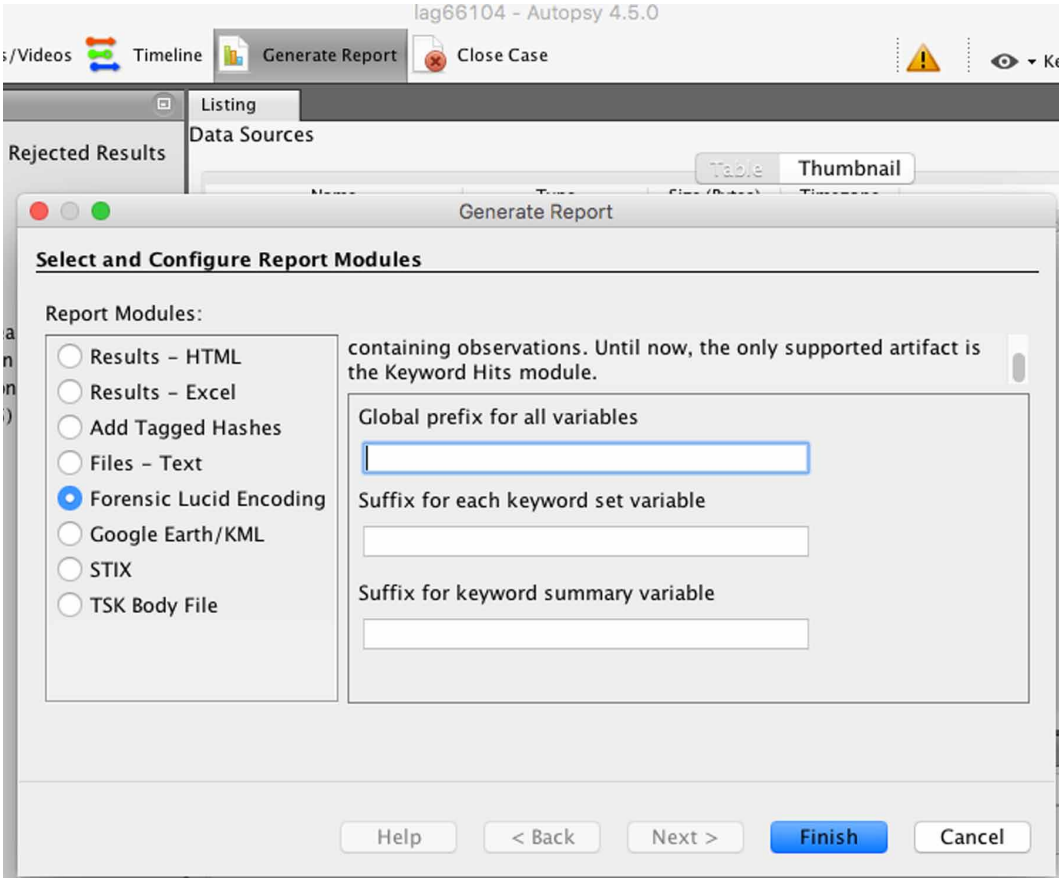
`<global prefix><keyword list entry title><global suffix>`

Resulting in observation sequence Autopsy_BruteForceAuthentication_Entries = ...

As mentioned, each sequence of observations is based on a keyword list entry and contains a set of tuples representing a *tagged hit* as a map of four key/pair values (Figure 12).

An investigator would have to look into the following dimensions: the Match dimension that contains the exact occurrence that corresponds to the given regular expression; then, he/she would see this match in the context given by the Preview dimension; the SourceFile dimension contains the physical file where this occurrence was found; finally, for correlation purposes, the investigator can appeal to the contents of the Tags dimension which has been meaningfully elaborated to describe a relevant trait in the observation. These dimensions are also exploited bythe Autopsy's built-in reports. We suggest the use of the *Tags* dimension to pinpoint pertinent observations that can be correlated through a reasoning engine.

Figure 10a. Forensic Lucid encoding generation process for the autopsy's keyword search ingester

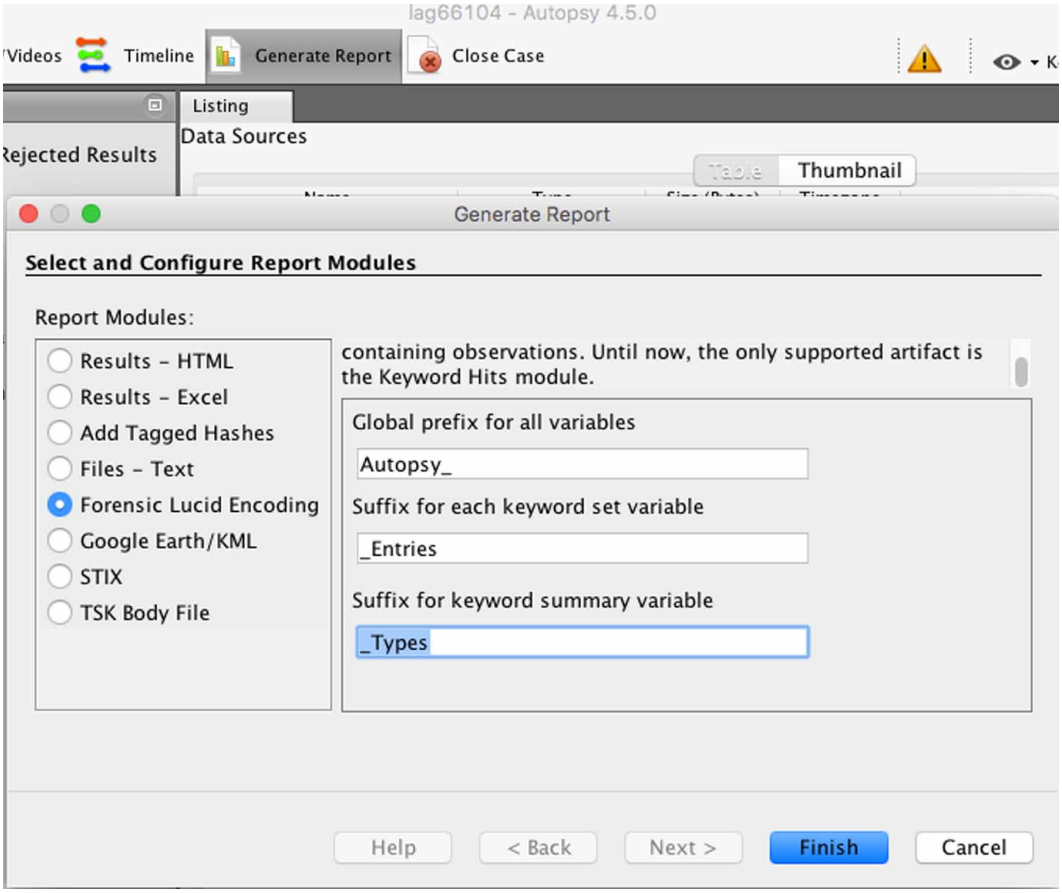



```
//Observations for the "Brute Force Authentication" set (Keyword Hits artifact).
observation sequence Autopsy_BruteForceAuthentication_Entries =
{
  (
    [
      Match: "login=badboy_2_envy&amp;passwd=",
      Preview: "er/pager2.shtml&amp;login=badboy_2_envy&amp;passwd=abc] for 50000 ms--",
      SourceFile: "/LogicalFileSet1/logs/audit_log",
      Tags: "HTTP_GET_REQUESTS_e1.bjs.yahoo.com"
    ],
    1,
    0,
    1.0
  ),
...
```

In case of no observations (the investigator did not tag anything), the Autopsy KeywordHits Types variable would simply hold the $ value: observation sequence Autopsy_KeywordHits_Types = $;

Regarding the credibility parameter, we established the subjective assumption that logs were generated under a controlled environment with no interruptions over time. Thus, under this supposition, a credibility value of 1.0 is given to all the observations for that source. However, as mentioned in Section 5.2, the Keyword Search ingester module has strong limitations regarding time-ordered observations.

Figure 10b. Forensic Lucid encoding generation process for the autopsy's keyword search ingester



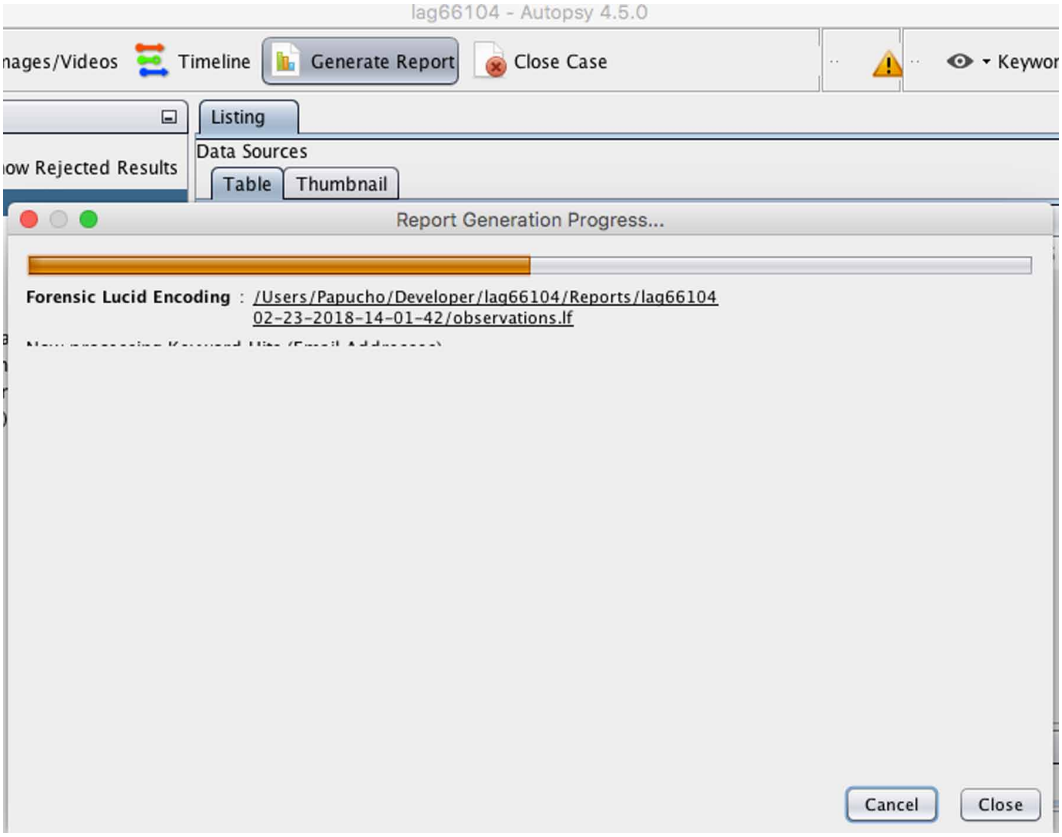
3.3. Volatility

In kernel mode, Windows has a layered abstract approach in handling the I/O Request Packets (IRPs), called device driver's stack (Ligh, Case, Levy, & Walters, 2014; Arasteh & Debbabi, 2007). Thus, several drivers can handle IRPs meant for a device even if that device does not belong to them. However, this feature could be misused by a malicious driver as follow. They would create a device of desired type and attach it at the bottom of another device driver's stack. Therefore, they would receive a copy of each IRP being sent to that *device driver* before the legit driver get them, and thus they could modify these IRPs according to their needs acting as a filter, as in the case of a Key Logger[13].

The device stack, sometimes referred to as device tree, could be shown from a memory dump using a plugin in Volatility Framework called *devicetree* plugin. That would help the investigator to inspect each tree (stack) and look if a malicious driver has attached its device to another device. There is no a definite line to differentiate between a clean attached device and dangerous one. It is up to the forensic analyst to look for the attached device type and the driver that owns it. In our case as will be seen later, the driver's name is "MRxNet", which is known as part of the Stuxnet Rootkit.

Next, we show how we modified the *devicetree plugin* to output the results in a Forensic Lucid Format to help the investigators in building his forensic analysis report.

Figure 10c. Forensic Lucid encoding generation process for the autopsy's keyword search ingester



3.3.1. Development Environment

Our Volatility development environment uses the Volatility last release 2.6 under Linux Ubuntu 16.04 x64 and Python version 2.7, and Sublime Text 3 editor. Windows could be used but since Volatility is a command line base tool, we have chosen to work in Ubuntu for efficacy and brevity.

3.3.2. Plugin Development Process and Usage Volatility installation:

1. Use a Linux Ubuntu machine version 16.04 or higher. Download Volatility source files (Volatility-2.6.zip) from http://www.volatilityfoundation.org/ releases or get the source files from github using:

```
$ sudo git clone https://github.com/volatilityfoundation/volatility
```

2. One way to do the installation is by creating a folder named according to the version to be installed such as *volatility2.6* into the user profile home directory. Volatility source files will be installed into the chosen directory and an upgrade later will be easier as well as having / running multiple versions of volatility in the same time. Extract the downloaded zip file into the previous folder named *volatility2.6*.

Figure 10d. Forensic Lucid encoding generation process for the autopsy's keyword search ingester

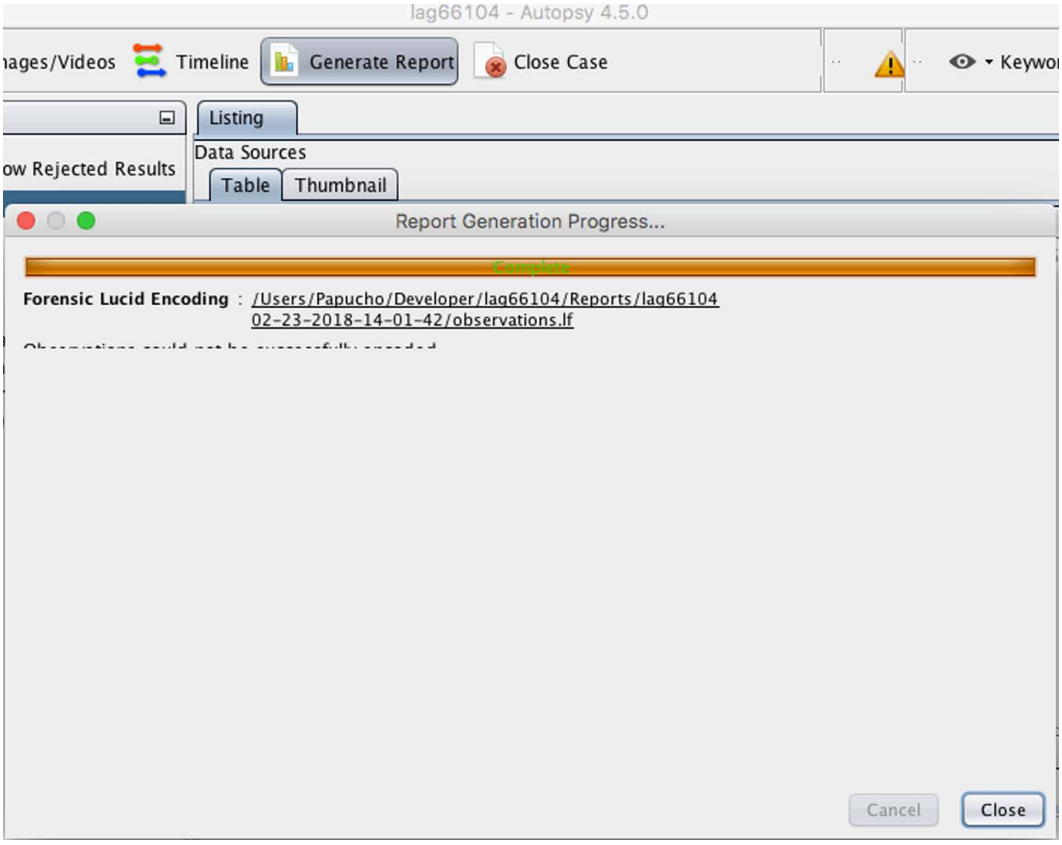


Figure 10e. Forensic Lucid encoding generation process for the autopsy's keyword search ingester

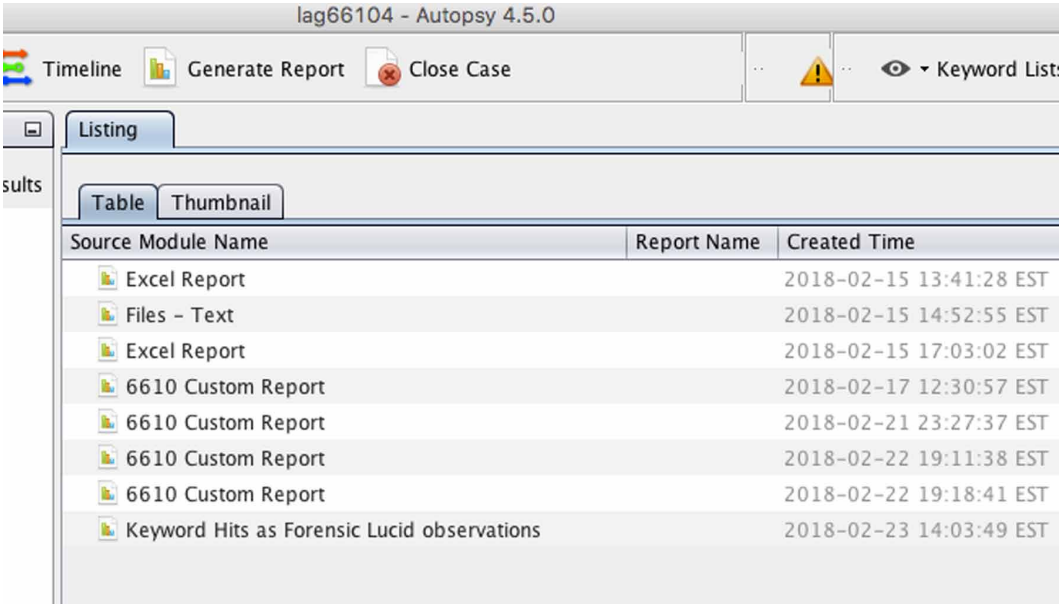


Figure 11. Tagged results under the Tags node

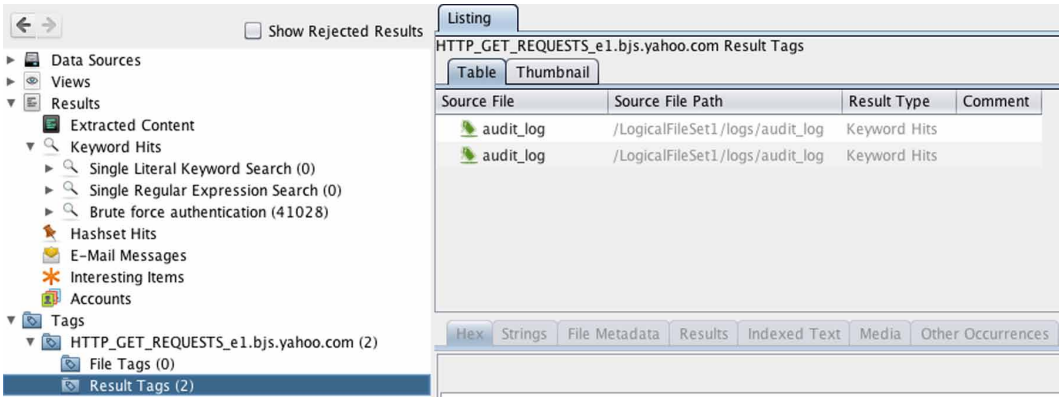
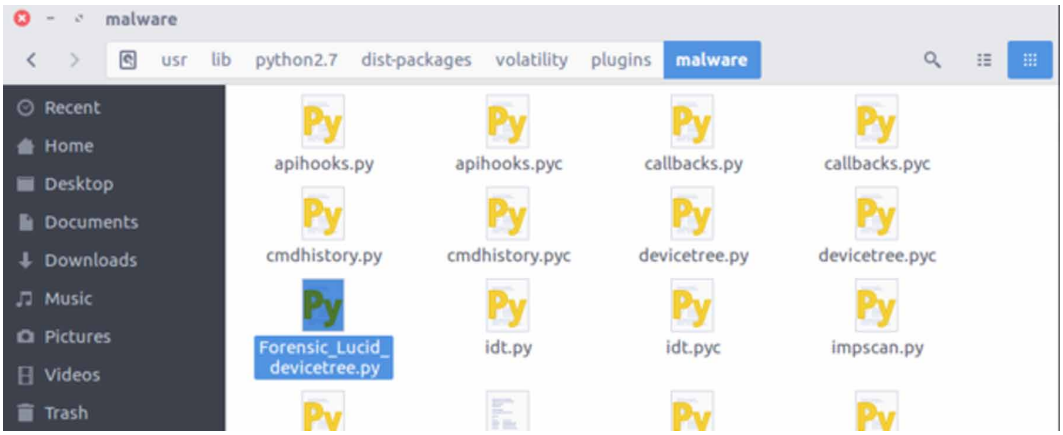


Figure 12. Observations for the "brute force authentication" set



3. Install dependencies. For Linux installation it is required other prerequisite packages / libraries such as:

```
$ sudo apt-get install pcregrep libpcre++-dev python-dev --y
```

4. For a comprehensive plugin support install the following libraries: Distorm3 (dependant plugin is apihooks), Yara (dependant plugin is yarascan), PyCrypto, PIL, OpenPyxl, ujason[14].

Plugin installation: Adding the new plug-in developed for Volatility is done as follows:

1. We assume that the forensic user has a Linux Ubuntu machine version 16.04 or higher with an IP address available for connections through SCP, SSH clients.
2. Identify the plugin python file named *forensicluciddevicetree.py* and copy the file into the Ubuntu file system using an SCP client. Copy the file anywhere into the local profile of your default Ubuntu user (for this tutorial the default user path is /home/osboxes)

3. Copy the file forensicluciddevicetree.py from the default location into the Volatility folder designated for malware plugins called malware as into the Figure 12. The path is: usr/lib/python2.7/dist-packages/volatility/plugins/malware/ or your custom Volatility installation folder:

```
#sudo cp /home/osboxes/ forensicluciddevicetree.py
/usr/lib/python2.7/dist-packages/volatility/plugins/malware/.
```

4. To make sure the plugin is successfully added to the framework run the following commands and verify that there is no error displayed and the name of the plugin is displayed as into the Figures 14 and 15). In case of displayed errors they may be related to python syntax errors or to dependency modules and functions imported / called from the python file:

```
$ sudo su
# python vol.py -h
```

Report generation process (how to use the plugin)

1. Copy a memory image/memory dump such as *stuxnet.vmem* or *zeus.vmem* into a folder belonging to default Ubuntu user or any other Ubuntu user.
2. Generate a new report that is exported into a text file with the extension *ipl* by calling the Volatility command line as illustrated in Figure 16.

Figure 13. Visualization of the new plugin, placed in the Volatility destination folder

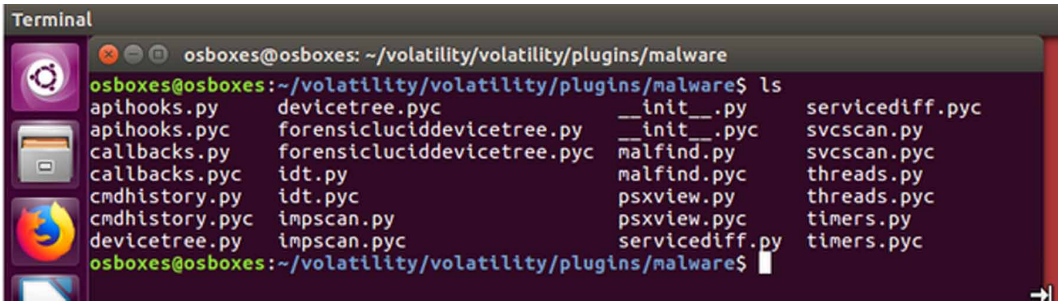


Figure 14. Visualization of the new plugin with its description using the help command line (I)

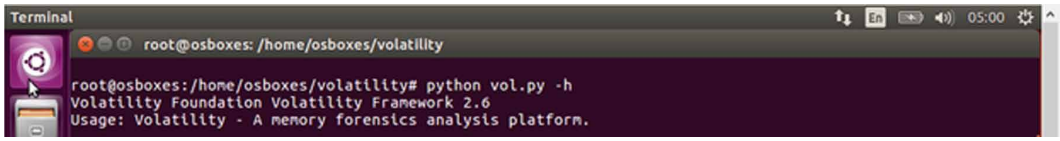
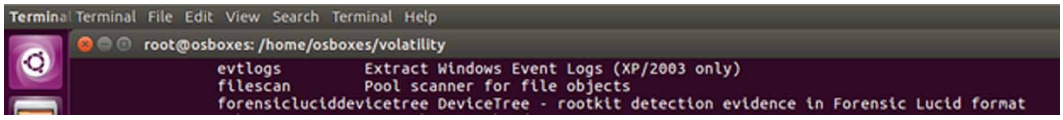


Figure 15. Visualization of the new plugin with its description using the help command line (II)



3. Inspect the output of the plugin to make sure it is working as expected, as illustrated in (Figure 17)
4. Generate a Volatility report that is exported into a text file with the extension *ipl* as illustrated in Figure 17. Other options/ formats of the output are dot, html, Json, sqllite, and xlsx.

Plugin development The meta/generic steps of writing a new plugin for Volatility are the following:

1. Install Volatility Framework.
2. Create a new file with a name. Note, this name will be the one used upon calling the new plugin in the command line.
3. Write the desired code in Python.
4. Copy the file to Plugins directory. Doing so, there will be no need to specify the path in the command line. However, this step could be skipped and therefore the absolute path of the plugin must be specified in the command line whenever it is used.

A customized volatility plugin may output its results in various formats (i.e. renderers) for example: text, dot, html, Json, sqllite, and xlsx. In volatility there is the concept of Unified Output which means that we write the code to generate the data one time, and then we output the data in multiple formats via renderers[15]. The output format for our plugin is going to be crafted as an output of a text renderer which is a normal text-based table output. An example of using a text renderer within Volatility is the following:

Figure 16. Generation of Output using the new plugin

Figure 17. Visualization of the new plugin output in forensic lucid format

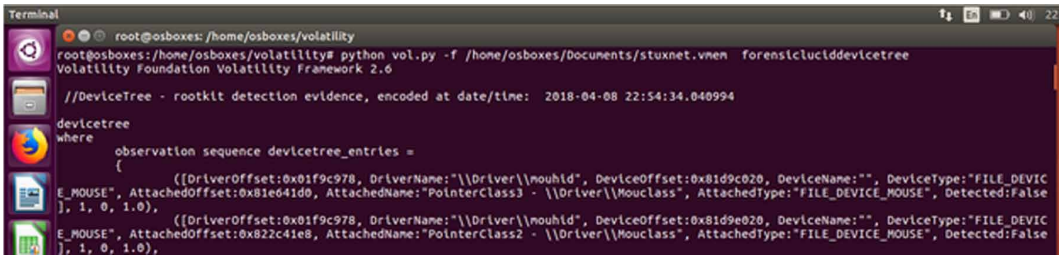
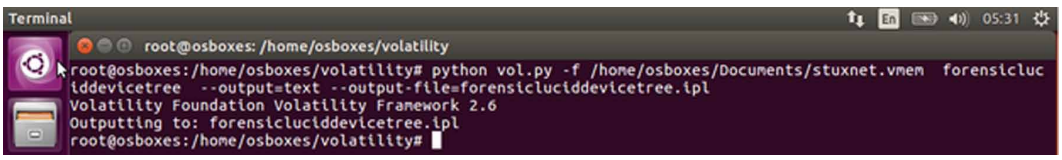


Figure 18. Exportation of the output into an external text file in forensic lucid format



```
$ python vol.py -f stuxnet.vmem plugin --output=text --output-
file=plugin.text
```

We are aiming to represent the results of the devicetree plugin in Forensic Lucid Format. To do so, we did the following:

1. We tried to understand what the devicetree plugin does by analyzing its code and use it to see its output. Also, in Volatility, one can show any specific options of any plugin by simply add in the code the argument: *-h* as following:

```
$> python vol.py <plugin-name> -h
```

The devicetree plugin is one of several plugins that are designed specifically to hunt rootkits and more general Volatility helps in hunting malware. Those plugins are located in a folder called /plugins/malware.

We used the devicetree to show information about the driver chain from a memory dump. The chosen memory dump (stuxnet.vmem) is infected with Stuxnet rootkit that installs a driver called *MRxNet.sys* into the kernel memory. A snippet of the result is shown in Figure 19:

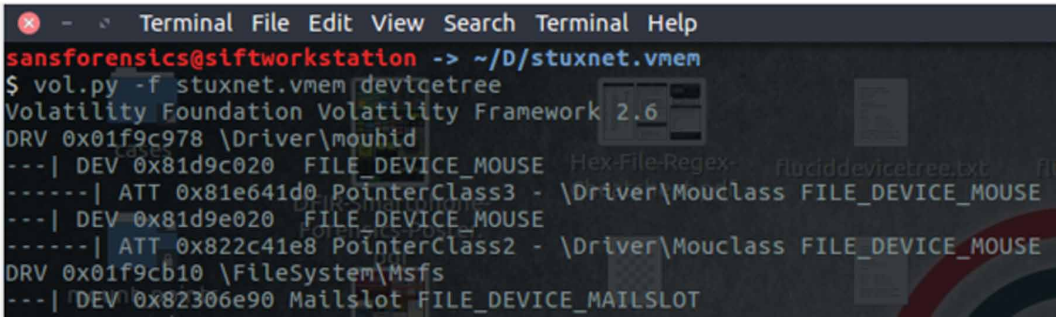
2. Each plugin is a python class. The name of the class becomes the name of the plugin. So, we created a file and named it *forensicluciddevicetree*, and then copied and pasted the original code of *devicetree* plugin. We have added new methods required to match the signature of the rootkit into the output using regular expressions. The new class is named *forensicluciddevicetree*, and the method render_text() is modified to present the results in Forensic Lucid Format. The created file is then saved and copied to the directory:

```
usr/lib/python2.7/dist-packages/volatility/plugins/malware/ as
mentioned into the plugin installation  paragraph.
```

The two main class methods found_rootkit_signature() and render_text() are presented into the UML class diagram in Figure 21.

3. In order to do the reformatting of the output, there is a standard method of every plugin called render_text() that needs to be customized for our own output objective. def render_text(self, outfd, data) The method parameters represent:
   (a) *Self*, refers to the instance of the class being used.
   (b) *Outfd*, stands for Output File Descriptor, output stream to which volatility will write. It is defined by the –output-file parameter of the Volatility command line .

Figure 19. Visualization of the memory image used as data source for the plugin



(c) *Data*, contains the data returned by the class method calculate() . In our plugin we do not define our own calculate() method but inheriting it from the superclass, in our case the *filescan.DriverScan*.

4. In order to search and detect the signature of the malicious rootkit we created a method called found_rootkit_signature (self, line). The method parameters represent:
   (a) Self, refers to the instance of the class being used.
   (b) Line, is a string type variable used to search for the know rootkit signature.
5. We have reused and renamed the original *devicetree* plugin classes named _DRIVER_OBJECT and _DEVICE_OBJECT. The first class returns a (Python) generator of Driver Objects that is the Driver's stack/tree chained by the operating system, the second class returns a generator of Device Objects that is a Device's tree/stack chained by the operating system. See their class diagrams into the Figures 20 and 21.
6. Test the new plugin by calling it in the command line as shown into the previous paragraph called Report Generation Process.

3.3.3 Generation of the Forensic Lucid observations

As mentioned earlier, one of the techniques used by stealth malware is hooking the IRPs by inserting a created device in the stack of other drivers at kernel-level. Therefore, the investigators would scan the whole drivers' hierarchy in order to identify any malicious drivers. This could be exhausting if the list is very long; hence, usually most targeted drivers are scanned first, those including: NTFS, FAT, Keyboard Driver, and TCP/IP drivers. Once a malicious driver is found, this is considered as an evidence and will be used in building the case by the investigation agency. Here comes the *Forensic Lucid* format to benefit from its features such as expressiveness. As into the Figure 17 and 19, the output of the *devicetree* is shown in two different ways: the default one and the one in Forensic Lucid format. Here is one of the observations in *devicetree* raw output format; the observation is related to the Device Driver at address 0x020d2f38:

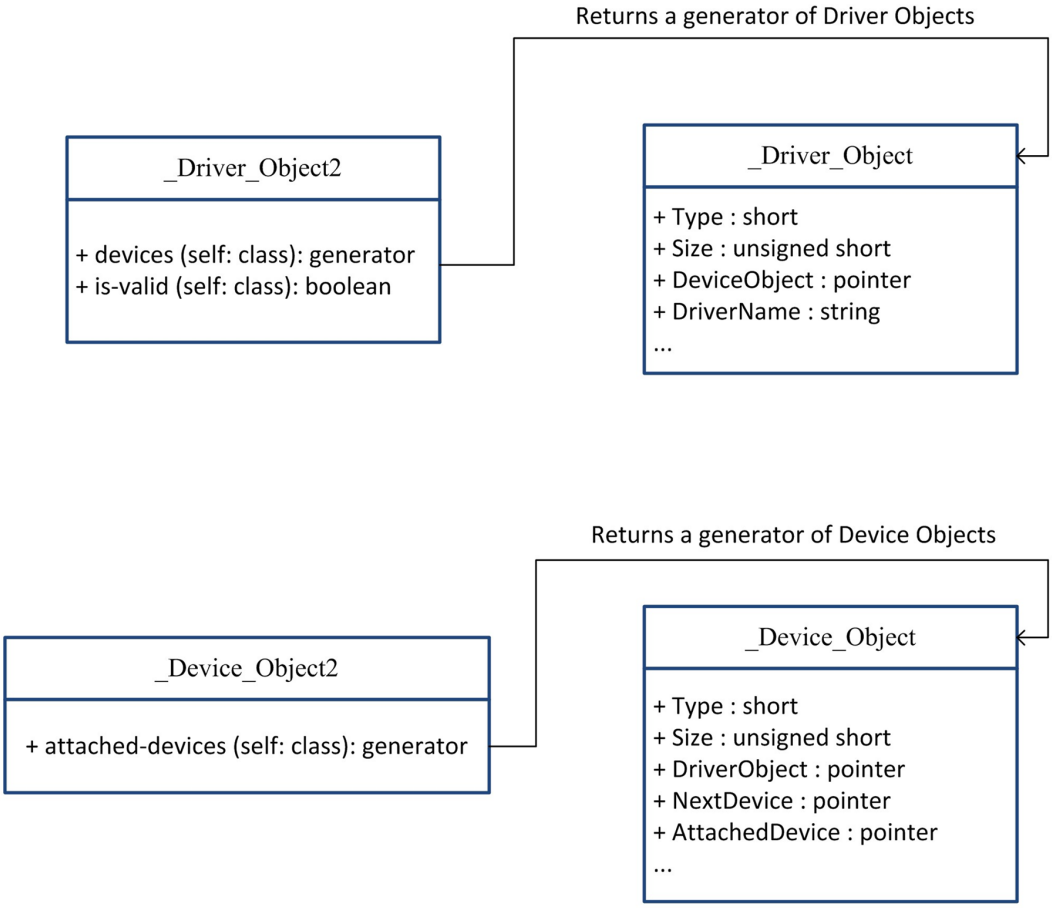
```
---| DEV 0x81e859c8   FILE DEVICE DISK FILE   SYSTEM
------| ATT 0x81f0ab90   - \ Driver \MRxNet_FILE DEVICE DISK FILE SYSTEM
---| DEV 0 x 81 fac 548   FILE DEVICE CD ROM FILE_SYSTEM
------| ATT 0 x 8226 ef 10   - \ Driver \MRxNet_FILE DEVICE CD ROM FILE SYSTEM
---| DEV 0x81f5d020   FILE DEVICE NETWORK FILE SYSTEM
------| ATT 0x821354b8   - \ Driver \MRxNet FILE DEVICE NETWORK FILESYSTEM
---| DEV 0x81bf1020   FILE DEVICE NETWORK FILE SYSTEM
------| ATT 0x81f0fc58   - \ Driver \MRxNet_FILE DEVICE NETWORK FILESYSTEM
---| DEV 0x82135d10   FILE DEVICE NETWORK FILE SYSTEM
------| ATT 0x81c0a910   - \ Driver \MRxNet FILE DEVICE NETWORK FILESYSTEM
```

Here are two observations depicted randomly from the observation sequence related to a Device Driver at memory address 0x020d2f38; The full output of the customized plugin is referenced in Appendix A.1:

Let's define each of the dimensions presented in the Listing 3:[16]:

1. *Driver Offset*: Is the memory address as a hexadecimal number where the device driver is loaded into the operating system and it is linked with the *Driver Object*.
2. *Driver Name*: Is the property called *Driver Name* in the operating system object called
   a. *Driver Object*.
3. *Device Offset*: Is the memory address as a hexadecimal number where the device is loaded.
4. *Device Name*: Is the name of the device computed by the plugin at run time.

Figure 20. Volatility plugin class diagram using UML notations (I)



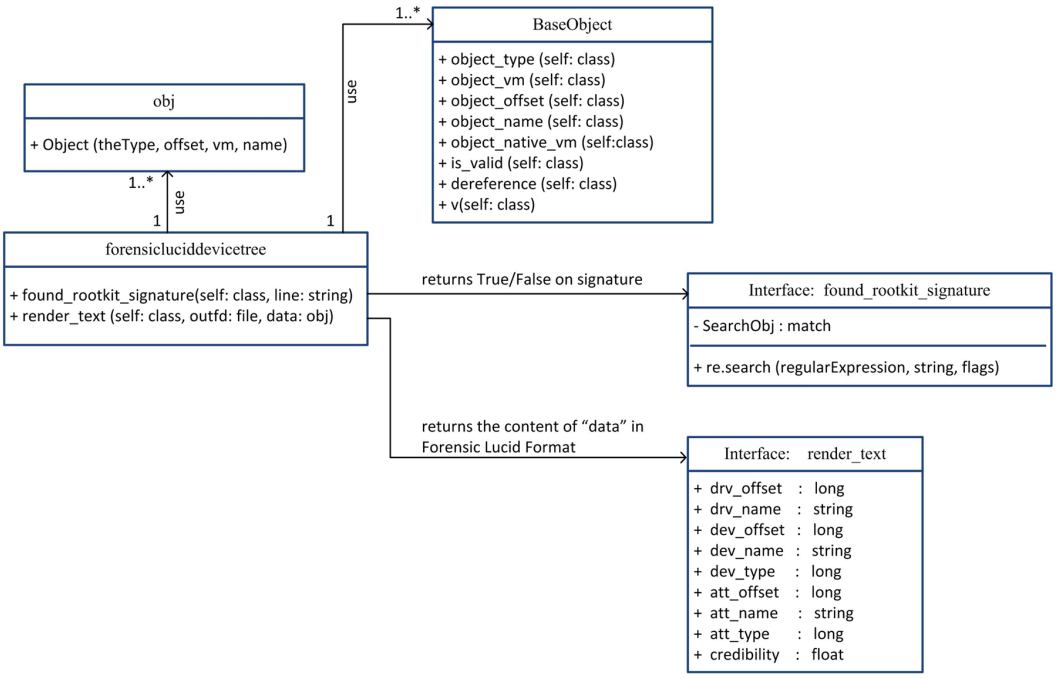
5. *Attached Offset*: Is the memory address as a hexadecimal number where the attached device is loaded.
6. *Attached Name*: Is the name of the attached device computed by the plugin at run time.
7. *Detected*: This is a property with two possible values True, False that indicates the detection of the rootkit signature into the current observation.

The dimensions that are important for the investigator are the following: *Drive Name, Device Type, and Attached Device Name and Detected*. The values of the first three dimensions may contain the trace of the rootkit attack and the fourth dimension is the flag for the detection of an attack. The

Listing 2. Standard Devicetree plugin raw output

```
DRV 0x020d2f38  \FileSystem \FltMgr
---| DEV  0 x8206b628    FILE DEVICE CD ROM FILE  SYSTEM
---| DEV  0 x81ead318    FILE DEVICE DISK FILE  SYSTEM
---| DEV 0 x 81 f 47020    FILE DEVICE DISK FILE  SYSTEM
------| ATT 0 x 81 fb9680   - \ Driver \MRxNet FILE DEVICE DISK FILE SYSTEM
```

Figure 21. Volatility plugin class diagram using UML notations (II)



Listing 3. Devicetree output in Forensic Lucid format

```
devicetree
where
  observation sequence devicetree entries =
  {([DriverOffset:0x020d2f38, DriverName:"\\FileSystem\\FltMgr",
    DeviceOffset:0x81f47020, DeviceName:"", DeviceType:"
    FILE DEVICE DISK FILE SYSTEM", AttachedOffset:0x81fb9680, AttachedName
    :"-\\Driver\\MRxNet", AttachedType:"FILE DEVICE DISK FILE_SYSTEM", Detected :
    True], 1, 0,1.0),
  ([DriverOffset:0x020d2f38, DriverName:"\\FileSystem\\FltMgr", Device
    Offset:0x81e859c8, DeviceName:"", DeviceType:"
    FILE DEVICE DISK FILE SYSTEM", AttachedOffset:0x81f0ab90,
    AttachedName
    :"-\\Driver\\MRxNet", AttachedType:"FILE DEVICE DISK FILE_SYSTEM",
    Detected:True], 1, 0, 1.0) };
```

forensic investigator will benefit from the Output File in Forensic Lucid Format by following the observations that have the dimension *Detected* equal to *True*; this indicate the presence of a STUXNET cyber-attack.

As forensic investigator, we would look first at the type of the attached device and the driver that owns this attached device. In the above example, the stack of *vmhgfs* (VMware Host to Guest File System) driver is shown. At the bottom of it, there is an attached device with no name (denoted by "-", of type: "FILE_DEVICE_NETWORK_FILE_SYSTEM" and is created and owned by a driver called *"MRxNet"*, which is known as a malicious driver. Any IRPs meant for the device HGFS, would be seen first by *MRxNet* and *MRxnet* has the option to play with these IRPs in its benefits. How *MRxNet*

does govern or filter these IRPs is found in the code that can be dumped out of the memory for static analysis using other Volatility plugins (Ligh et al., 2014).

It is important to note that *devicetree* plugin is not used alone in hunting for malware. There are multiple possibilities to do correlations of evidence obtained from other plugins such as *ldrmodules, idt, gdt*.

For the calculation of the value of *Detected* dimension we are using the class method called found_rootkit_signature() that we apply for every observation from the sequence . We are using a regular expression in Python to match the substring "MRx" in the following instances collected from the *devicetree* output:

1. *DeviceName*: ''MRxClsDvX''
2. *DriverName*: ''\Driver\MRxCls''
3. *DriverName*: ''\FileSystem\MRxDAV''
4. *DriverName*: ''\FileSystem\MRxSmb''
5. *DriverName*: ''\Driver\MRxNet''
6. *AttachedName*: ''- \Driver\MRxNet''

We assigned a reliability of 1.0 for every observation output that contains the dimension *Driver Offset*. The credibility needs to be calculated using reliabilities of all witness accounts as per Dempster's Rule of Combination (Mokhov, 2013). In our case we have only one witness account which is *devicetree* entry/output corresponding to a specific *Driver Offset*, therefore the uncertainty value of every individual observation is set at 1.0.

4. EVALUATION AND RESULTS

4.1 Achievements

Autopsy The construction of a *module* under the Netbeans' RCP platform was possible to take advantage of this extension mechanisms as well as its integration into the report engine to interact with the *Keyword Search ingester* module. In this way, any data source can be used as input and each *tagged keyword hit* in it is treated as a relevant observation when the report is generated.

Additionally, we attained the successful compilation and verification of the syntax of the Autopsy's Forensic Lucid plugin output using the following *gipc.jar* command line:

```
#java -jar gipc.jar --flucid observations.ipl
```

Volatility We have achieved our goal on providing a variant of the standard plugin (*device-tree*)[17] of Volatility tool, plugin that can be used in dead forensic analysis to extract evidential information and encode it in Forensic Lucid Format. We have introduced a property called "Detected" to help identifying a possible sequence of events that prove a malware is present into the operating system. Our Volatility plugin has been tested and it is scalable to data sources (e.g., memory images) collected from different operating systems.

The successful compilation for the verification of the syntax of the Volatility Forensic Lucid output was done using the *gipc.jar* command line:

```
# java -jar gipc.jar --flucid forensicluciddevicetree.ipl
```

4.2. Important files

Table 1 shows the links to the team's important files at the repo.

5. CONCLUSION

We successfully implemented the proof-of-concept plugins for Autopsy and Volatility for sample cases and their encoding into Forensic Lucid format. The latter format is essential for forensic case knowledge representation and reasoning in uncertainty that is formally specified and sound. We are working to release our code and data open source.

5.1. Evidence correlation

Identify Brute Force Authentication attack' procedure

The following procedure will be done by a forensic specialist using Autopsy forensic tool.

1. Use a standard keyword list in Autopsy called IP Addresses to generate a search of all IP addresses from the open proxy log data source.
2. Define a new keyword list as a regular expression for the login attempts (using HTTP GET requests into the access.log log file) Use the expression "login=.*passwd=". Trigger a search for the new defined keyword list and observe the Results section.
3. Continue the correlation using only the access.log file. Define a new TAG called Brute Force Authentication. This will be used in tagging both the results in IP Addresses keyword hits and Brute Force Authentication keyword hits.
4. Look into the Brute Force Authentication results, choose at random a *login and password* item from the list and open the access.log indexed text where the login attempt is highlighted in yellow color. Pickup up manually into a notepad editor the IP address that appears into the HTTP GET Request for the respective login and password. Open the IP Addresses keyword hits results and go at the marked down IP Address. Once into the access.log indexed text, the IP addresses will be highlighted in yellow color, count how many logins attempts the source IP address has done. If we have many or a significant group of logins attempts on different yahoo web sites around the world, it means that the Attacker attempts to spread his brute force attack in order to not have his IP address blocked by an IDS, IPS or a Firewall. TAG into the Brute Force Authentication results the previously chosen login and password item with the TAG = Brute Force Authentication.
5. Once detected the malicious behaviour by repeatedly login attempts at previous step, TAG the Result IP Address with the TAG = Brute Force Authentication. The IP address is the source IP address of the offender /attacker /hacker unless the attacker is spoofing his source IP address.
6. Repeat the steps 4 and 5 for as many IP addresses the forensic specialist wants and is able to extract from the logs.
7. Prepare to generate a Forensic Lucid report that has the IP address of the offender and the login and password on the (yahoo) website that is the target of the attacks. Using the new report module *Forensic Lucid Encoding*, chose from the tagged results the TAG = Brute Force Authentication and start the *Forensic Lucid Encoding* report generation. Export the output report file and add it to the Investigation Report. The report will be used to prepare the Suggested Accusations.

Table 1. Team's important files from repo

Type	Resource
Volatility's plugin output	flucid/Volatility-output/ForensicLucidDeviceTree.ipl
Volatility's source code	Volatility/Plugin%20Source%20Code/
Volatility's OS object types	volatility/plugins/overlays/windows/xp sp2 x86 vtypes.py
Autopsy's source code	autopsy/lfplugin/
Autopsy's binary plugin	autopsy/ca-concordia-c6610.nbm

5.2. Limitations and Future Works

Autopsy When using the Autopsy tool, we searched for evidence of brute force authentication attacks. The observations extracted from the data source file audit log are not all evidence of brute force authentication, that is to say, the most of them are legit traffic. Each observation from our report contains a form of authentication that includes passing the username/password credentials as part of the URL requested within an HTTP GET request. As a future work, semi-automatic correlations of source IP addresses of the request that outline login attempts from HTTP GET/POST requests, could be made in order to establish an evidence for brute force authentication. See the appendix 5.1 for the proposed procedure.

As some limitations, we noticed that:

1. The Preview dimension into the standard Excel report shows a limited amount of characters, so we cannot extract important data such as the full timestamp or the full HTTP request that was generated from the attacker's machine. To support this, the keyword ingester module would need to implement regular expression groups.
2. The observations in the default keyword ingester module are stored without time information which is critical to the formalization of evidence (order and gaps in the observations are not discernible).
3. On the Keyword Search results tab, when used for analysing log files and searching for three category of attacks such as IRC Connections, Spamming, and Click-thru fraud, a very limited amount of keyword hits were found while the Indexed Text node showed a range of pages with multiple matches per page. In consequence, these searches were discarded since they were not consistent. In the Brute Force Authentication searches, however, we have not found such evident inconsistencies. We think that this behavior affects the trustworthiness of the tool with respect to the data source. A further analysis might be conducted to establish in a statistical manner how accurate the search capability is, making an observation's credibility value more realistic.

These limitations will affect the reconstruction of the event based on the log analysis method using the standard keyword ingester module since they are being transferred and mirrored into our Forensic Lucid Encoding report module.

Volatility Our volatility plugin is scalable to memory images collected from various operating systems. We have tested the plugin against multiple memory images and we generated the reports in *Forensic Lucid Format*. The additional memory images used for testing are infected with malware such as Zeus.

The limitation for the Volatility plugin is that our regular expression used for the search is limited to one string associated with the STUXNET rootkit and as future work the list of the strings can be enhanced with additional strings (i.e., signatures represented as regular expressions). The additional strings would be added to the class method called found_rootkit_signature ().

REFERENCES

Arasteh, A. R., & Debbabi, M. (2007, September). Forensic memory analysis: From stack and code to execution history. *Digital Investigation Journal*, *4*(1), 114–125. doi:10.1016/j.diin.2007.06.010

Carrier, B. D. (n.d.). Autopsy forensic browser. *Sleuth Kit.* Retrieved from http://www.sleuthkit.org/autopsy/

Carrier, B. D. (n.d.). The Sleuth Kit. *Sleuth Kit.* Retrieved from ‚http://www.sleuthkit.org/sleuthkit/

Ding, Y. (2004). *Automated translation between graphical and textual representations of intentional programs in the GIPSY* [Master's thesis]. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Retrieved from http://newton.cs.concordia.ca/_paquet/filetransfer/publications/theses/DingYiminMSc2004.pdf

Edwards, S. A., Townsend, R., & Kim, M. A. (2017). Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE international conference on formal methods and models for system design, MEMOCODE 2017*, Vienna, Austria, September 29-October 02 (pp. 175–184). Academic Press. doi:10.1145/3127041.3127055

Han, B. (2010). *Towards a multi-tier runtime system for GIPSY* [Master's thesis]. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

Honeynet Project. (n.d.). *Honeynet forensics project scans.* Retrieved from http://old.honeynet.org/scans)

Ji, Y. (2011). *Scalability evaluation of the GIPSY runtime system (Master's thesis)*. Montreal, Canada: Department of Computer Science and Software Engineering, Concordia University. Retrieved from http://spectrum.library.concordia.ca/7152/

Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). *The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons.

Mokhov, S. A. (2013). *Intensional cyberforensics [PhD thesis]. Department of Computer Science and Software Engineering, Concordia University*, Montreal, Canada. Retrieved from http://arxiv.org/abs/1312.0466

Mokhov, S. A., Assels, M. J., Paquet, J., & Debbabi, M. (2014, November 3–5). Automating MAC spoofer evidence gathering and encoding for investigations. In F. Cuppens et al. (Eds.), *Proceedings of the 7th international symposium on foundations & practice of security (fps'14)* (pp. 168–183). Springer. 11 doi:10.1007/978-3-319-17040-4_11

Palmer, G. (Ed.). (2001). *A road map for digital forensic research, report from first digital forensic research workshop*. DFRWS.

Paquet, J. (2009, July 20–24). Distributed educative execution of hybrid intensional programs. In *Proceedings of the 33rd annual IEEE international computer software and applications conference (COMPSAC'09)* (pp. 218–224). IEEE Computer Society. doi:10.1109/ COMPSAC.2009.137

Paquet, J., & Kropf, P. G. (2000). The GIPSY architecture. In P. G. Kropf, G. Babin, J. Plaice, & H. Unger (Eds.), Proceedings of distributed computing on the web (Vol. 1830, pp. 144–153). Springer Berlin Heidelberg. 17 doi:10.1007/3-540-45111-0_17

Pourteymour, A. H. (2008). *Comparative study of Demand Migration Framework implementation using JMS and Jini [Master's thesis]*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Retrieved from http://spectrum.library .concordia.ca/975918/

Tao, L. (2004). *Intensional value warehouse and garbage collection in the GIPSY [Master's thesis]*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Retrieved from http://spectrum.library.concordia.ca/8129/

Tong, X. (2008). *Design and implementation of context calculus in the GIPSY [Master's thesis]*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Retrieved from http://spectrum.library.concordia.ca/975704/

Vassev, E., & Paquet, J. (2008, April). Towards autonomic GIPSY. In *Proceedings of the fifth IEEE workshop on engineering of autonomic and autonomous systems (EASE 2008)* (pp. 25–34). IEEE Computer Society. doi:10.1109/EASe.2008.9

Vassev, E. I. (2005). *General architecture for demand migration in the GIPSY demand-driven execution engine [Master's thesis]*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Retrieved from http://spectrum.library.concordia .ca/8681/

Wu, A. H. (2009). *OO-IP hybrid language design and a framework approach to the GIPC [PhD thesis]*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Retrieved from http://spectrum.library.concordia.ca/976610/

ENDNOTES

1 https://github.com/volatilityfoundation/volatility/wiki
2 http://www.volatilityfoundation.org/releases
3 https://github.com/volatilityfoundation/volatility/wiki/Installation
4 https://github.com/volatilityfoundation/volatility/wiki
5 https://code.google.com/archive/p/Volatility/wikis/CommandReferenceMal22.wiki
6 https://securityintelligence.com/zeus-analysis-memory-forensics-via-Volatility
7 https://netbeans.org/features/platform
8 http://www.sleuthkit.org/autopsy/docs/api-docs/4.5.0/moddevpage.html
9 http://www.sleuthkit.org/autopsy/docs/api-docs/4.5.0/modreportpage.html
10 https://platform.netbeans.org/tutorials/nbm-google.html
11 http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/i18n-branding.html
12 Ryan C. Barnett. Scan 31. Retrieved from http://old.honeynet.org/scans/scan31/sol
13 https://code.google.com/archive/p/volatility/wikis/CommandReferenceMal22.wiki#apihooks
14 https://github.com/volatilityfoundation/volatility/wiki/Installation
15 https://github.com/volatilityfoundation/volatility/wiki/Unified-Output
16 https://github.com/volatilityfoundation/volatility/blob/5dfdff5570c6d9d6cd92c4f6debbeee002b2edea/volatility/plugins/overlays/windows
17 https://code.google.com/archive/p/volatility/wikis/CommandReferenceMal22.wiki#apihooks

Carlos Rivas is a software engineer with experience in consulting for telecom and banking companies. His main professional interests are about enterprise software architectures and information systems security.

Cristian Stanciu has been working in the field of IT since 2005 as a Senior Information Technology Consultant for the private company Hypertec-Group Montreal with focus on network architecture and design, network security, virtualization and Linux-based systems. He holds IT certifications with Cisco (CCNP-security, CCNP routing), CompTIA (Linux+) and VMware (VCP-DCV). As academic education, Cristian has a Bachelor of Science in Computers awarded by Technical University of Cluj-Napoca, Romania, Faculty of Automatics and Computers, 1991-1996 and a Master of Engineering in Information Systems Security awarded by Concordia University Montreal, 2016 -2018.

Zahra Zohoorsaadat has completed a Master of Information Systems Security at Concordia University. Her fields of interest are Digital Crime and Forensics, and Network Security.

Serguei Mokhov had obtained his PhD from Concordia University, Montreal, Quebec, Canada, where he completed his bachelor's and master's degrees in Computer Science and Information Systems Security. Mokhov's diverse research interests include intentional programming, distributed and autonomic computing, digital forensics, information systems security, AI, software engineering, computer graphics and HCI, Linux, and computer networks. His PhD dissertation had to do with intentional cyberforensics. He also teaches at Concordia and Champlain College of Vermont various subjects in those disciplines. Dr. Mokhov works together with Drs. Joey Paquet and Mourad Debbabi on Forensic Lucid, General Intentional Programming System, MARF and its applications, web services, and cyberforensic analysis.

Dr. Miao Song on the interdisciplinary Illimitable Space System (ISS) for real-time interactive graphics for performance arts and interactive documentary having received coverage in Montreal and abroad.