Student Number: 100780122
Name: Joaquin, Moreno Garijo

Mac OS X Forensics

Supervisor: Dr. Lorenzo Cavallaro

Submitted as part of the requirements for the award of the
MSc in Information Security at
Royal Holloway, University of London

I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished works of other people. I declare that I have also read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signature:

Date:

Academic Year 2013/2014

Abstract

Mac OS X Forensics

by
Joaquin Moreno Garijo

Master of Science
in
Information Security

Royal Holloway, University of London
2013/2014

During the past few years, the number of incidents related with Mac OS X environment have increased, especially those related to malware. Due to this new tendency, incident response teams and computer forensics investigators require new tools, procedures and documentation to understand how to properly handle the incidents over this platform.

However, the research undertaken till now is not enough to deal with these new threats. For this reason, the research community has started to work on volatile evidences, live system evidences and malware reversing approaches. However, the research done over persistent evidences has almost been non-existent, in particular the evidences created by Mac OS X itself.

Hence, this dissertation focuses on Mac OS X persistent evidences, identifying the evidences, how these evidences are stored and how this information can be obtained. Furthermore, the project provides an implementation that extracts the information stored by Mac OS X using a well-known forensics framework called Plaso extending the tool and allowing for the work done in the dissertation to be used by the computer forensics community at large.

Acknowledgement

This thesis would have not been possible without Kristinn Gudjonsson. Thank you.

I want to thank Lorenzo Cavallaro to gave me freedom during the project to take decision related with the implementation and the goals of the project.

Lastly, I would like to thank to Plaso team that provided this incredible framework where on top of it, I developed the Mac OS X plugins extending the tool functionalities.

Contents

List of Figures

List of Tables

Hardware is easy to protect: lock it in a room, chain it to a desk, or buy a spare.
Information poses more of a problem. It can exist in more than one place;
be transported halfway across the planet in seconds; and be stolen without your knowledge.

Bruce Schneier

1 Introduction

Information security is a broad subject with different specializations and roles. As a previous experience in the roles related with incident handling, incident response and computer forensics, the author took the decision to focus the dissertation in a topic related with of these areas, especially in that ones related with computer forensics and operation systems. During the research process in computer forensics topic, the state of art revealed that several investigation were done over Linux and Windows environments [23, 22, 60]. Also, the research community has worked over new operation systems such as IOS or Android due to the increased demand over these technologies [46, 45].

However, the research done over Mac Os X environment is insufficient taking into account the increased number of malware that have been found since 2012 [54]. Due to this lack of tools, documentation and research to analyse incidents over the Mac OS X platform, the author took the decision to focus the project on this platform and the evidences created by this operation system.

1.1 Aims and Objectives

Mac OS X represents the 7.54% of the world desktop operation systems market share [66], the 14.04% in the United States and the 6.23% in Europe [87]. Despite the shared market, the research in the field of computer forensics and malware was not developed. It was believed that Mac OS X was not the target from potential intruders or malware. However, in 2012 the Flashfake botnet infected around 600.000 Mac OS X computer [54]. This trend has been continued in 2013 and 2014 with other Mac OS X malware focus on exploit Java and Office vulnerabilities [20].

Due to this necessity to develop a Mac OS X forensics tools, different companies have been researching and implementing solutions over this operation system [33] without published the process to extract and analyse these forensics evidences. Also, different governments and researchers are working in the extraction process of evidences, as an example in the early 2013 the Defense Cyber Warfare Technology Center from South Korea [57] have been published a Keychains extraction paper. Related to the open source

community, some tools have been developed in the context of memory, such as Volatility [91] or Volafox [48] and other related with live system evidence [89]. However, any forensics tool has developed that can extract inside the file system evidences (persistent evidences) due to the lack of documentation, the binary format that most of the evidences are stored and the fact that most of these evidences are unique for Mac OS X. The objective in this document is identifying the Mac OS X persistence evidences, provided a technical explanation about how these evidences can be extracted.

Hence, the dissertation focuses on the most important evidences created by Mac OS X. It explains where the operation system stores these evidences, what kind of information they contain, how the information can be extracted, and finally, providing an implementation that permits the extraction of these evidences where the source code is referenced in Appendix A and B.

The evidences were classified in two main groups: the ones that contains timestamp values and without timestamp (no time valuable). Relating to the timestamp evidences, the dissertation explains the Apple System Log in BSD and Binary format (6.1), Basic Security Module (6.2), the most relevant Property List (6.4), CUPS IPP control files (6.6), UTMPX (6.3), Keychains (6.8) and so on. Whereas in case of not time valuable evidences, the dissertation explains the user system accounts (5.2), the autologin functionality (5.3), the binary recent files attribute bookmark (5.4), persistence and boot time executable (5.6), etc.

1.2 Implementation and Availability

For each evidences documented in the following chapters, a parser has implemented being able to extract the information contained by the evidence. Due to the source lines of code written during the dissertation, around 15.000 lines, all the code is available on the Internet instead of being included in the present document. The implementation has done in two steps.

During the first step a set of debuggers independent parsers have been developed. These parsers extract the data stored by the evidences providing the object relative positions, structure type, data type and steps done during the analysis and extraction process helping to understand how the information was stored by the operation system. The main aim of these parsers is to help to understand how the information is extracted and documented in the following chapters. These parsers are stored in the project Mac OS X Forensics in the URL *code.google.com/p/mac-osx-forensics/* and documented in Appendix A.

Related to the second step, the forensics framework Plaso has used [42]. The tool extracts the timestamp evidences from different type of hard disk images, mounted partition or individual evidences. Despite it supports different evidences from Windows and Linux environments; no implementation has done over Mac OS X. This dissertation has used the framework to extend the functionality of the tool providing the capacity to extract the evidences created by Mac OS X. The implemented parsers are available since *Plaso 1.1* released on *6th of June 2014*. The project URL is *plaso.kiddaland.net*,

the source code is available in *code.google.com/p/plaso/* and documented in Appendix B.

In all the implemented parsers, the author made a reference in the code indicating that it is a dissertation part from the M.Sc. Information Security at Royal Holloway University of London providing the author's university email as a contact.

1.3 Structure of Report

We begin in Chapter 2 explaining the most important terminology related to digital forensics backgrounds and the state of art in Mac OS X showing the lack of documentation and implementation in operation system evidences. Having understand the basic concepts and the actual drawbacks in Mac OS X forensics, a presentation of the forensics framework Plaso is done in Chapter 3, that has used to implement the dissertation plugins that will extract the evidences. In chapter 4, Mac OS X is explained from a digital forensics perspective: based on a combination of operation system, different timestamps and different manners to store the timestamp depending on the evidences, unique binary files evidences, etc. Hence, the different evidences that can be extracted from the file system will be explained. In Chapter 5 we focus on an evidences that are important for forensics purpose, but they are not time measurable, whereas in Chapter 6 we focus on evidences that can be ordered by time (timestamp evidences). Additionally, the dissertation has been accepted to the RootedCon security conference in March 2014 [74]. The conference indicates that the work is an initial research of the M.Sc. dissertation at Royal Holloway providing the supervisor and author's name.

Regarding to the appendix A, it contains the available Python's source code of independent parsers that extract the evidences from the most important Mac OS X binary artifacts [88]. They have been published as proofs of concept without require the Plaso framework. These plugins have been provided to the forensic community to be evaluated and checked against real incidents. The appendix B has the Plaso plugins implemented for the dissertation providing the reference and the URL to download the source code. In both appendixes the source code is not added due to the fact that it has been implemented more than fifteen thousand code lines providing an online access to this code. In all of these plugins and tools, the author uses his name and provided as a contact the Royal Holloway email address indicating that it is a part of M.Sc. dissertation at Royal Holloway. The new Plaso plugins are public available since version 1.1, published on 6th June of 2014. The appendix C will explain all the BSM token structures documented in Chapter 6.

If you reveal your secrets to the wind, you should not blame the wind for revealing them to the trees.
Kahlil Gibran

2 Digital Forensics Background

During this chapter the main concepts related with forensics methodology and state of art focus on Mac OS X will be explained. First of all, the basic concepts about incident handling and forensics methodology are described to understand the goals that the project will archive. Secondly, the author will expose the research that has been done in Mac OS X until now, which parts might require an additional work and in which specific part the project will be focused: Mac OS X persistent file system evidences.

2.1 Forensics Methodology

Computer forensics is a discipline focus on obtaining evidences that provides a clue about how, when and who did these actions over a computer. Computer forensics is a step inside the incident handler methodology. It is necessary to understand how an incident is managed to realise why a computer forensics might be required during an incident. Incident Handler is divided in six different steps [37, 31, 80, 50]:

1. **Preparation**: during this phase the incident handler team needs to accomplish all the previous works before an incident. The goal is to be ready and to avoid errors during a real incident: policy, procedures, law enforcement, checklists, management support, emergency plan, prepared team / tiger team (training process must be done), jump bag or toolkits, point of contact, etc.

2. **Identification**: identification of abnormal behaviour that can identify an alert. This step is done analysing the events, correlated them and looking for deviations in the normal behaviour. An incident must be declared if the deviation has likelihood to have harm objective. It requires a clear, easy and understandable checklist, done during the preparation phase, that first levels can apply without required the implications of higher levels, and also, avoiding actions than can altered the evidences.

3. **Containment**: prompts actions to prevent than the incident can affect more assets. In this step is where the forensics process takes place. It is recommended

to obtaining the evidences firstly avoiding that the countermeasures or corrective action can affect or altered the evidences; however, and due to external reason such as economic implications or the necessity to continue delivering a service, where some times the preventive actions are done before the forensics evidences acquisition.

4. **Eradication or remediation**: it is considered a slow process. It requires deleting all the artifacts from the assets that were affected during the incident. The common approach is restore the assets to the previous state applying all the measures to avoid that the incident can be repeated: change passwords, applied updates, restore from backups, build a new system, fix custom vulnerabilities, etc. Sometimes the forensics process can be done in this step; it depends on the priorities of the owner from the affected asset.

5. **Recovery or confirmation**: it is the process where the affected assets come back into production. During this process the assets are checked and monitored with special care, looking for something suspicious to be sure that the eradication or remediation step has done properly.

6. **Lessons learned or aftermath**: in this process all the parts implicated the incident: administrators, incident handlers, security management, etc. must meet together to analyse the reasons of the incident, if it can be avoided, apply new countermeasures, update procedures, policies and guidelines, reporting errors during the incident and the measures required to avoid that it might happen in the future.

As it was describe before, forensics is a corrective action that take place inside the containment step in the incident handler methodology. Computer forensics might be required to archive correctly the further steps during an incident. The investigators must know how the incident happens to be sure that they have applied correctly the measures during the eradication or remediation step. A computer forensics investigation can provide the clue about how the incident was emerged, and then, provided enough information to applied correctly the eradication or remediation measures, an also, help during the aftermath step to understand witch measures must be applied in the future: technical measures, change policies, update the business continuity or recovery plan, law enforcement, etc.

Understanding the different steps related to the forensics methodology, the initial required research over the Mac OS X forensics could be achieved. During this process, the persistent evidences created by the operating system have an important weight during a computer forensics case. The forensics methodology is divided in a few initial steps based on the evidence acquisition, then, cyclic steps during the process of identify, analyse and extract the results from the evidences, and finally, a report with the obtained results must be written [79]:

1. **Previous information**: the forensics analyser needs to know the role of the affected host, the authorised users, the software installed, the place in the networks

where it was installed and all the actions done during the identification phase. The role of the preparation and identification phase has a very important weight, because a bad preparation and identification phase can destroy useful evidences or the incident can be detected so late.

2. **Evidence acquisition**: during this step the goal is to acquire the evidences that will be used during the forensics case. First of all, the volatile evidences need to be acquired before the system is halted avoiding that the evidences will be lose. Each order executed in any server must be store in a notebook indicating the command executed, who did the action, in witch host and the time where the action was done. These evidences can be the list of process, listening services, active users, mounted file system, active connections, etc. Moreover, the acquisition of the memory has an important impact due that it provides many useful evidences, especially against malware. After acquiring the volatility evidences, the physical evidences must be obtained: hard disk, flash memory, USB, etc.

   All the acquired evidences are known as a best evidences and never need to be used during the next steps. A copy of each of these evidences must be done and work only with these copies. Also, the forensics analyser must use hash algorithm to have assurance that the original evidence and the copy are exactly the same. Additionally, the original copy must be save in a secure place that prevents access to unauthorised user, water flood, fire, impact or another possible issue.

   For each evidences, the investigator take notes about when, who and what they do during the evidence acquisition. This information must be store in a paper called *chain of custody*. All the times that the evidences is moved to other place or they are in contact with other person must be reflected in this document. In the *chain of custody*, the auditors must note when the evidences were acquired (time), how were acquired (process), who was the responsible (person) and in case that the evidence is a digital evidence the hash algorithm must be calculated plus the number of device where this evidence is saved (hard disk identification). The document must be signed for the analyser that acquires the evidence or the evidence responsible during the process between the place where the evidence was acquired and the secure room.

3. **Extracting the timeline evidences and artifacts**: get all the events or actions from the evidences that can be ordered by time creating a timeline. Also, the forensics investigators can use an specific evidences that despite they do not have a timestamp valuable can provided useful information such as software installed, software executing during the next boot time, mounted devices, previous open documents, etc.

4. **Analysing the results**: this step is the most artesian process and required a previous experience. The forensic analyser needs to understand the events and the implications of the events: what event is considered a normal event and what is suspicious depends on the case. Sometimes one event alone does not reveal an important threat, whereas if it is a combination with other events, it can represent an alert or incident.

5. **Analysis the suspicious action**: the suspicious action must be analysed to confirm that it was a malicious action or only a false positive. During this step the forensics requires to focus on new timeline points that it was no considered before, and even need to do a handwrite parsers to some evidences depending of the case, coming back to the previous process extracting a new timeline. Also, strings, signatures or values can be identified where a byte or string search against the evidence might be required.

6. **Extract the evidence**: the evidences that can be considered as a clue to the case are extracted and stored in a specific place separately from the working evidences. If it is a malware artifact, usually it is provided to the reverse engineer malware team to analyse the artifact to know the functionalities and goals of that malware. The last four steps are a cycle step process until the forensic case was accomplished.

7. **Report**: a final report should be written indicating all the information obtained during the case. This report only reflects the obtained evidences without providing an opinion or conjecture.

In the following subchapter the different types of forensics evidences will be explained, understanding the aims that they achieve and the research done over this type of evidences over Mac OS X environments.

2.2 Volatile Evidences

Volatile evidences are the group of evidences that exists temporally in a host, that if the system is halted, these evidences are lost. There are three main volatile evidences: memory (RAM), live system evidences and network evidences.

The live system evidences are checked for the first-level incident response group who has a list of steps to get relevant information such as users log on, execute processes, connections or mounted devices [23] that permit the detection of anomalies that should be reported to the second level. The first level must write which action and when was executed. To avoid as much as possible that these actions modify the file system, they must be done using a static binaries from an external device using as less as possible libraries and programs from the compromise host [38]. Furthermore, these evidences never have to be stored in the compromise file system. Usually the auditors use pipe network tools, such as NetCat or SSLCat, to store the output results of these tools in an external host instead of the local host. The main disadvantage of this methodology is, that despite using static compile tools and the output is sent to an external host, a little bit of modifications is done in the file system [38]. Additionally, some malware and attackers have implemented modules that detects these kind of software applying antiforensics mechanism that are able to delete the most important evidences, altered the file systems, and times providing a wrong results.

Related to the network evidences, it is always interesting to store the network traffic from the suspected host to known what kind of communication is done between the

target host and other hosts [82]. Usually two approaches are recommended: using a Test Access Ports (TAPs) that can read the traffic between two points or using a mirror port from a switch or router that permits the traffic retransmission from a specific port to the mirror port [36]. Usually this traffic is stored in a Packet Capture file format (PCAP) to be easily analysed offline by tools such as Wireshark or Tshark.

Finally, the volatility memory evidence, usually associate with the RAM, has acquired an important relevance in the past years due to the research process in this kind of evidence [38]. The capacity have changed from only be able to extract the strings stored in the memory to be allowed to extract the list of process, process dump (without the import table), network communication, listen ports, active users, process remove from the list of process and other useful information [92].

In case of Mac OS X, the tools that permit the extraction and processing these evidences are described below, where most of them have been developed during the last two years:

- Live System: the binary tools can be created from Darwin ports or mac ports using static compilation (creation by hand). In September 2013, the developer *sud0man* published the tool called Pac4Mac [89] that can extract useful information from a Mac OS X File System. However the tool uses binaries from the live operation system instead of implement its own parsers having a high impact modification access time in the files system. For this reason, the tools must only be used in a read only file system evidence against an acquired file system and never against a real live file system.

- Traffic Network: we can detect traffic anomalies or well know attacks using software intrusion detection system (IDS) such as Snort and Suricata or hardware IDS solution using Tcpreplay or Scapy to recreate network traffic from a stored PCAP file. To read and analyse the PCAP file the tools Wireshark, Tshark, Tcpflow or Tcpdump can be used [51].

- Memory: to extract the information store in the memory the recommended tool is the Mac Memory Reader from ATC-NY [29] that has full support to 32 and 64 bits. To analyse the extracted memory, the tool Volatility framework [91] provide a big number of Mac OS X plugins to extract useful information from the memory. Also, Volafox [48] has useful plugins. Furthermore, the Mandiant company provided the tool called Mac Memoryze [61] that can extract and process the memory. Some investigators prefer to use Volatility whereas others prefer to use Memoryze.

2.3 Malware

During a forensics analyser, the auditors might find a specific piece of software that can be installed in the system with illegitimate purpose. This software is called malware and should be analyse to understand the behaviour and the goals or reason about why this piece of software is installed in the system. Usually, the companies have a specialised

team to do this analyse and it is considered as a sub specialisation in the computer forensics world.

The malware analyse can be done using two different approaches or the combination between both: behavioural analysis (fast analysis) and code analysis (slower and deep analysis) [83]. The behaviour analysis is done when the incident response group requires the first approach and initial information about the malware. The process consists to execute the malware in a controlled environment, commonly a virtualization environment, where usually two hosts are used: the host where the malware is executed called **infected host** and the **trap host**.

The operation system from the **trap host** should be different than the infected host. The goal of the trap host is to deceive the network communication infected host using tools such as Honeyd, Trapd, Fakedns and services like IRCd, Apache and Bind. The Linux distribution RemNux created by Lenny Zeltser is a Linux distribution that provides all this functionalities [97]. Additionally, in the infected host a set of programs must be installed to log the acceded, deleted, modified and created files during the execution of the malware. Also, it is helpful having a network tool that show the communication. Regarding to code analysis, the investigators use three types of tools: unpacked tools, debuggers tools and disassembled tools.

Mac OS does not have the same amount of analysed malware tools than other operation system like Windows, but it still has a few basics tools. The most common tool for behaviour analysis is *Fslogger*: File System Change Logger [85] with the GUI interface *FSEventer* [73] (Figure 2.1). The tool stores a log indicating with files and directories were created, modify and delete; however, as Dino explains in Black Hat 09 conference [98], this program can be jumped by sophisticated malware. Regarding to network monitor, Apple provided the tool called *Activity Monitor*. Related to code analysis, and due to the fact that IDA Pro [83] has Mac OS X support, malware analysers have the required tools to make a full code analysis. An example of behaviour and code analyse from the Mac OS X malware's Dockster.A can be found in the Mila Parkour web page called Contagio Dump [71].

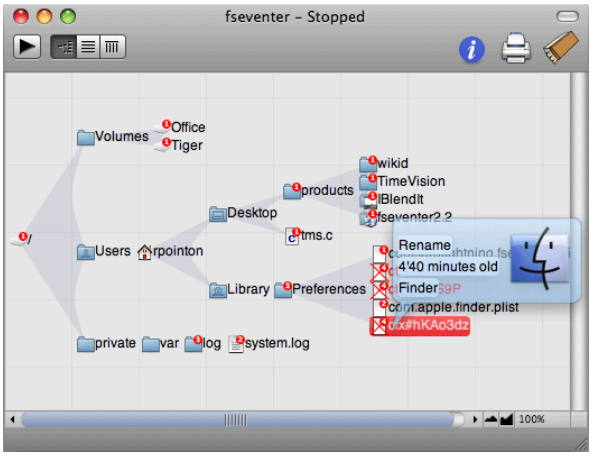

Figure 2.1: FSEventer screenshot from the Dockster.A [71].

2.4 Persistent Evidences

The persistent evidences are the evidences that still existing in a host despite the system is shutdown, in other words, the evidences that can be found in the file system. The majority of the persistent evidences can be ordered by time line, where auditors can reconstruct the actions during a specific time period. The union between all the timestamp evidences coming from the file system, artifacts and logs is called *Super Timeline* [30]. The persistent evidences can be separated in two main types: the file system itself and the artifacts plus logs:

- The file system stores timestamp attributes from files and directories. Usually, for each directory or file the MAC time is provided: Modification time, Access time and Creation time [21]. In case of Mac OS X, the file system used is Hierarchical File System Plus (HFS+) where additional information is added for each entry (discussed in chapter 4).

  For each different time assigned (MAC times) in each directory and file, an entry in the time line must be created. As an example, for a file "X" three different entries are created: one from the time when the metadata of the file was modified, other from the last time the data file was modify and finally from the last time that the file was acceded. Some file systems add the timestamp when the file was created, as an example NTFS or HFS whereas others when the file was deleted like EXT3. In case of Mac OS X, the tool The Sleuth Kit can extract these timestamp from the HFS filesystem where the tool called *Mactime* can ordered the timestamp [52].

- The second group is the artifacts [78] and logs. They can be separated in two groups: the evidences that have a timestamp value and the group of evidences that have not timestamp values. The logs, by definition, have timestamp values, whereas the artifacts not always it is possible to obtain a timestamp value, despite they can still provide useful information. As an example, an operation systems like Mac OS X, store in the file system useful information such as execute processes, application that will be launched during the boot time, user account or application installed where, despite that they do not have timestamp value, it provided useful information.

  The most common Mac OS X artifacts and logs are the security audit event provided by Basic Security Module (BSM), system event and applications logs stored by the Apple System Log (ASL) and independent log files, configuration files usually stored using Property List, the keychains file to store the passwords, binary files such as UTMPX or CUPS IPP that provide an extra information, etc. These evidences are explained in the following chapters.

Hence, Mac OS X has a big number of evidences, most of them not documented, that are unique from this operation system (except IOS) such as Plist, Keychains, ASL, Cache, DocumentRevision, BSM, etc. where the number of forensics tools that are able to extract these evidences have practically not being developed. Moreover, it was not until 2012 when Mac OS X was publicly considered a target from malware and APTs [54]

[47]. Due to this new tendency, some companies have developed functionalities to extract Mac OS X evidences, but without technical explanation about how this information is extracted. As an example, in the 2013 CEIC conference Guidance Software have done a presentation about its new support in Mac OS X [33] indicating how the information can be extracted using EnCase product.

During the next chapters, the document focuses on the acquisition of these persistent evidences, explaining where this information is stored, how it is stored, how it can be read and finally an implementation will be provided to extract this information, generating a valid Super Timeline where the forensics framework Plaso will be used as a core of the project.

I don't always create new tools, but when I do, I name them all using Icelandic acronyms.

Kristinn Gudjonsson

3 Plaso

Plaso is an open source forensics framework implemented in Python that provides an unique tool to parse computer forensics artifacts, log files and file system timestamp generating an unique output where the timestamp from all the evidences are correlated in the same time base, doing the daily job easily for computer forensics investigators [42].

During this chapter the author will introduce Plaso, how to extract the evidence required as an input for the tool, the basic orders to use the tool, the different interfaces that the tool provides, and finally, explaining the advantages and disadvantages to implement the Mac OS X parsers over Plaso framework and the implemented parser list developed during the dissertation.

3.1 Introduction to Plaso

In 2010, Kristinn Gudjonsson wrote the SANS Gold Paper releasing the tool called log2timeline [43]. The idea was provide a tool that is able to extract the timeline from different application artifacts evidences. At the same time, Joachim Metz released different libraries to work with some forensics artifacts such as Windows Shadow Copies [62] or Internet Explorer history format (MSIE Cache File) [63] and other libraries required during a forensics incident like the library that permits work with FileVault2 cypher partitions [64].

Plaso (plaso langar a safna llu) was developed to provided a multithread framework environment that unify the log2timeline tool with the Joachim's libraries and Python interface for TheSleuthKit libraries, adding the capacity to Plaso to extract the file system timestamp. With this idea, Kristinn and Joachim have been working in a framework that can extract the file system timestamp, operation system artifacts/logs and the applications artifacts/logs. All the evidences are normalised in the same timestamp base generating a super time line that it can be correlated by the forensic investigators/analysts correctly and easily without requiring extra work [42].

Plaso works against a file system image or partition image without only required

specify the image, providing as an ouput all the evidences extracted from this image and ordering the extracted evidences in the same timestamp base. Furthermore, Plaso can works against a mounted partition or single evidence in a manner that can be used for a specific purpose or as an extra tool during an incident. Moreover, this can be done with only a few commands making easily the extraction evidence process providing more time to the investigator to work in the case itself instead of working in how the evidences can be extracted.

Plaso has being designed with a design that makes easy to work with different timestamps, unknown and well know binary evidences, text parsers with macros, SQL query engine, output formatters, etc. where a new plugin and functionalities can be easily developed and added to the framework.

At the end of 2013, Plaso was able to extract the majority of the file system timestamp including NTFS, EXT, HFS and FAT32. Also, it was able to extract a large number of applications evidences such as Chrome, Firefox, Safari, Xchat, Google Drive, Java IDX or Skype. Relating to operation system evidences, Plaso was able to extract the majority of Windows evidences such as VSS, registry, Windows logs, LNK, jobs, Prefetch, etc and a few from Linux such as SELinux or Syslog. However, no job was done over Mac OS X.

3.2 Working with Plaso

During this section, the author will explain the basic commands required to use Plaso as a forensics tool during an Mac OS X incident. The Plaso version used is 1.1. release candidate (RC1), where most of the evidences documented during the dissertation have already been implemented. At the beginning of June, the stable Plaso version 1.1 has been released being available for the forensics community.

3.2.1 Evidence acquisition

The investigators must obtain the file system evidence, called image, to be able to work with Plaso. However, the process of acquiring the evinces over Mac OS X is a little bit different that other classic environments such as Linux, Windows and virtual environments. For this reason, during this section the different steps to acquired the evidence will be explained:

- Obtaining the hard disk from the computer and make a bit-for-bit copy using a software or hardware solution. This approach is by far the best; however, the new Apple devices does not permit dismounting the hard disk since it is soldered to the motherboard avoiding the use of this technique.

- When the hard disk cannot be dismounted, one solution might be boot the Apple device in read only recovery console. To do this, the investigators must press the button combination *Cmd* + *s* during the boot process (EFI) [33]. When the

recovery mode has being started the auditor can make a software copy over an external device connected by an external port using the tool dd. As an example, to make a disk0 hard disk copy over the *mnt* partition the required order is as follow: *dd bs=512 conv=noerror,sync if=/dev/disk0 of=/mnt/mac_osx.dd*.

The disadvantages of this approach are several. First of all, the copy is done over an environment that can be altered by some malware or intruder. Also, the read only mode may create some alterations over the file system if it was not shutdown correctly [33]. Moreover, the recovery mode only accepts HFS partitions, and then, the external device must be formatted in HFS, and secondly and due to the read only mode, the auditor cannot create any directory in the system to mount the external device, and for this reason, in the above example the directory *mnt* was used (a empty directory must be used).

- The next solution must be applied when the had disk cannot be dismounted from the computer. It requires using an external cd-rom or dvd-rom device (read only device) with a Linux distribution and forensics tools. To select the Live CD the auditor can decide to use private solutions such as Encase or Helix or free solutions such as SANS SIFT, Kali, Backtrack or any live cd that at least has the DD, DC3DD, MD5, SHA and NetCat tools. To acquire the evidence, the auditor must turn on the Apple computer but booting from the live CD instead of the hard disk. To do this, the live cd must be introduced into the CD or DVD unit and booting the Apple computer pressing the letter *c*. When the Linux environment has being lunched, the auditor can use the tool DD or DC3DD to make a software copy over an external device or using the network. Taking into account that DC3DD must be used only with traditional disk and not with solid state disk.

  As an example, to make a disk0 hard disk copy over the *mnt* partition the order is as follow: *dd bs=512 conv=noerror,sync if=/dev/sda of=/mnt/image.dd* or using an network instead of a mounted external device *dd bs=512 conv=noerror,sync if=/dev/sda | nc external_ip external_port* and in the external computer *nc -l -p external_port >>image.raw*. Warning: the network traffic will be in clear.

3.2.2 Executing Plaso

Plaso provides different frontends to work over the input evidence depending of the deep level that the auditor wants to work [40]. The fronted *log2timeline* is the responsible to extract all the evidences from the forensics hard disk image: file system timestamp, operation system evidences and applications artifacts. To work with *log2timeline* first of all, the user needs to indicate from which partition from the hard disk he or she wants to get the timeline. The fronted provides an option that permits to list the available partitions without required using external tools such as *mmls (TheSleuthKit)*. This option is the *–partition_map* that prints the list of partitions from this hard disk evidence enumerating the partitions with a integer number.

Knowing the number of the partition, the auditor only needs to indicate the number of partition, the hard disk image and the name of the output file that contains all the

extracted evidences. As an example, if the acquired hard disk is called *disk_image*, the partition number is *5* and the output file name is *output_file* the two orders are as follow:

```
$ log2timeline.py --partition_map disk_image
$ log2timeline.py --partition 5 output_file disk_image
```

With these two orders, the auditor will have in the file *output_file* all the extracted evidences related with the partition *5*, ordered by time. This file is created with the Pstorage format (Plaso storage format). The fronted *Pinfo* works over this format that provides a resume of the number of evidences, the timestamp when this *output_file* was created and the type of evidences extracted. The tool only requires as an argument the *output_file*. As an example, in figure 3.1 the author has used Plaso over a Mac OS X environment where the Careto APT was executed. The partition analysed was the number 5 that contains the root partition. In Counter Information and Plugin Counter information *Pinfo* reveal different types of persistent evidences that they will be discussed in the next chapters.

```
$ pinfo.py output_file
```

```
        version = 1.1.0-dev_20140227
        cmd_line = /usr/local/bin/log2timeline.py --partition 5
careto.out /mnt/hgfs/moxilo/Desktop/careto.dd
        preprocess = True
        runtime = multi threaded
        method = imaged processed

Counter Information:
        Counter: total = 1372170
        Counter: filestat = 1364844
        Counter: syslog = 3725
        Counter: plist = 1032
        Counter: bsm_log = 987
        Counter: asl_log = 861
        Counter: sqlite = 694
        Counter: mac_securityd = 18
        Counter: mac_keychain = 6
        Counter: utmpx = 3

Plugin Counter Information:
        Counter: plist_default = 1013
        Counter: mackeeper_cache = 693
        Counter: safari_history = 7
        Counter: plist_install_history = 6
        Counter: plist_spotlight = 2
        Counter: plist_softwareupdate = 2
        Counter: ls_quarantine = 1
        Counter: plist_spotligth_volume = 1
        Counter: plist_macuser = 1
```

Figure 3.1: Pinfo over Mac OS X image with Careto malware executed.

Having extracted the timeline evidence with *log2timeline*, the *Psort* fronted can be used to work over the result file *output_file* permitting use some specific filters in SQL query format to indicate a range of times, types of evidences, text parsing, etc. An example, using *Psort* over a Mac OS X image to extract all the events between two dates printing the date, time and message fields is explained below [41]:

```
$ psort.py -q output_file "SELECT date,time,message WHERE date >
    '2014-04-01 22:00:00' AND date < '2014-04-02 10:30:00'"
```

The auditor can use also the fronted *Psort* to change the default Pstorage format to other exportable formats such as SQLite, Elastic search or CSV [39] because some investigators prefer to use GUI interfaces than an a basic console. The CSV format can be used to load the super timeline in any spreadsheet tool such as Excel or LibreOffice. The SQLite format can be used to be load in the forensics GUI tool called 4n6Time showed in figure 3.2. Also, Plaso permits to dump the data in Elastic Search format being able to load this output in the Kibana web interface. As an example, the figure 3.3 shows the Kibana interface after load an Elastic Search output from the Mac OS X raw image that was running the careto malware.

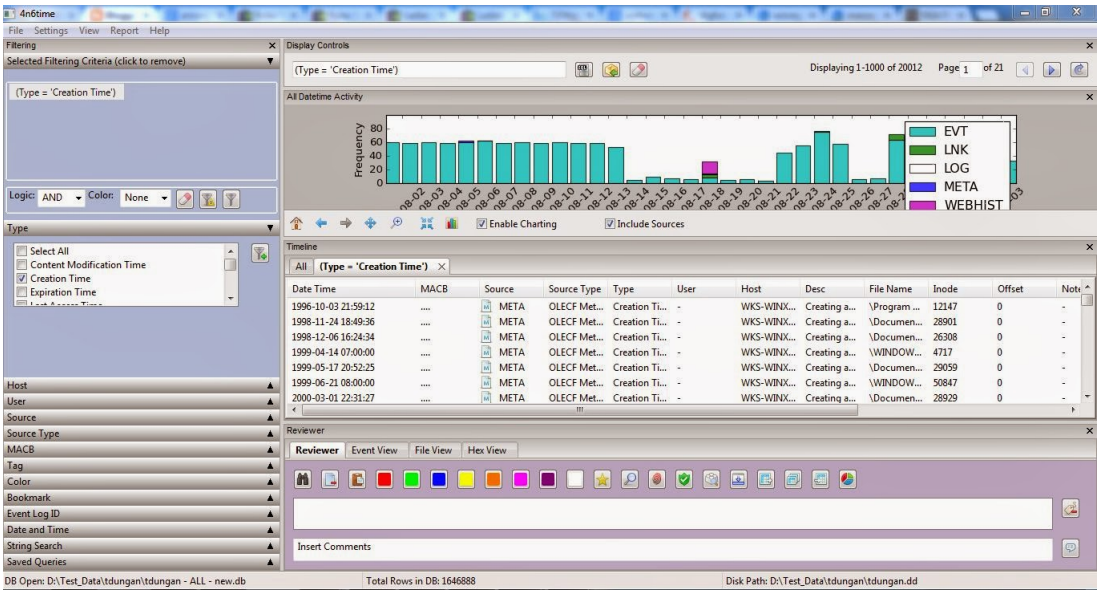

Figure 3.2: Example of 4n6Time tool obtained form Dav Nads blog.

Plaso has a fronted called *Plasm* that permits to specify some rules conditions over the timeline evidences to detect some knows behaviours. As an example, if the logs comes from the plaintext syslog or from the apple system log and the text contains *USBMSC* or *fsevents* an external device has been connected in the computer [44]. Plaso provides a default group of rules for different operation system. In case of Mac OS X, the default rules contains only a few cases, where after analyse different user behaviour and different malware, a new rule version has been implemented and documented in

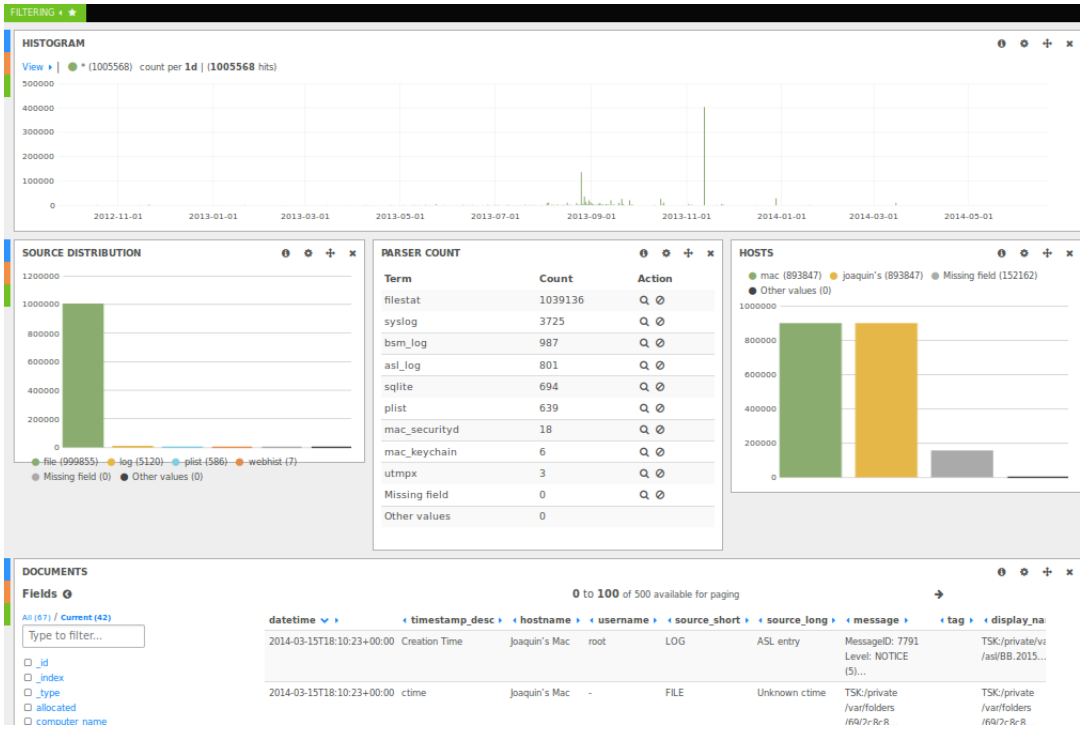

Figure 3.3: Kibana interface from Mac OS X image.

Appendix 3: Case of Study. The fronted *Plasm* only requires as an input option the *rules* files where the set of rules where defined and the *output file* created previously.

```
$ plasm.py -t rules output_file
```

3.3 Implementation process

The implementation of each evidence will be done in two difference steps. During the first step an independent parse will be programmed to analyse the evidence with several debuggers functions and methods that provided the required support to understand how the evidence is stored by Mac OS X. These files are being published as a "Mac OS X forensics" project in Google code [88] identifies the project as a part of the MSc project from the RHUL. Furthermore, and due to the independence of each parse, the forensics community will be able to use the parser again real case providing a feedback without the requirement of any specific framework or extra libraries.

Once the evidence is appropriately understood, its implementation will be coded using the forensics open source framework Plaso providing important benefits. First of all, the framework manages the timestamp issues that permit to work with different timestamps, formats and file types that Mac OS X has. Also, and due to Plaso is a well know tool, the implemented parsers are going to be used for a several forensics

professionals, where a system images from real cases will be used. Additionally, due to the code submit requirements over this project, the parsers will have better quality, performance improvement and understandable.

The drawbacks of use Plaso is principle the required entry level barrier to understand the framework and the extra time to implement the parser. First of all, for each implemented parser the developer must provide evidence with all the different cases that it might have and the parser should be able to extract. Then, a unit test against this evidence must be implemented to be sure that the parser works as expected if the core of the framework or the parser itself is modified. Also, to be able to submit a parser the code must be checked against the Pylint library to validate that the code follow the code-quality recommended by Python Foundation. When the parser follows the recommendations, the code is submitted to be reviewed using the Code Review web application. Then, one of the main code developers will review the code, indicating which parts are wrong for reason such as the code is not easily understandable, slow implementation, better options to implement it, some errors conditions that were not take into consideration during the developing process of the parser, characters codification issues and so on.

This process can required a few days to more than one month, when sometimes more than ten difference CLs (reviews) are required. When the review is finished, the code is submitted as a future release candidate version of Plaso. When the release date arrives, another core developer reviews the code and report to the developer issues that was not taken into account during the first revision. Finally, for each release candidate version, different environments are going to be used against the parser to validate that they work as expected without generating noise or false positives in other environments.

Due to this long process of review code, when a research from one evidence is done, the Plaso parser has being developed to be allowed to provide the full implementation over Plaso before the dissertation deadline. The developed parsers for Plaso are described in table 3.1 where the whole property list parser are described in table 3.2. All the implemented parsers provided during the dissertation are available since the Plaso version 1.1, released on seventh of June of 2014 and announce in the Plaso webpage project (figure 3.4) and documented in *Appendix B*.

For each plugin, Plaso divides the parser in different files with different objectives. Inside the directory *parsers* the parser itself is found. In the formatter directory, another file is created where the developer must define how the output information will be stored to the output file. The unitary test must be created in the same directory that the parser (*parsers*) with the extension *_test*. The test file may be moved to a new directory in future releases. Additionally, some evidences have required new timestamp methods to work with its timestamp or plaintext structures where the Plaso libraries such as *text_parser* or *timelib* have been modified. Finally, a new directory called *unix* has been created to define some maps from Mac OS X structures like Basic Security Module (BSM).

Plugin	Review	Commit	Codereview
UTMPX	2013-11-15	2013-11-27	26820043
Wifi	2013-12-01	2013-12-09	38160043
ASL	2013-12-06	2013-12-13	38170043
BSM	2013-12-10	2014-01-15	39820043
AppFirewall	2013-12-12	2014-01-01	39810044
Pyparsing single line log	2014-02-08	2014-07-21	57170053
Pyparsing BSD log	2013-12-24	-	41530045
Securityd	2013-12-25	2014-01-05	41530045
Cups	2014-01-05	2014-01-20	47920043
Document Revision	2014-01-20	2014-01-27	54790043
Configuration files	2014-01-14	2014-01-30	52450043
Keychains	2014-01-25	2014-02-07	57030043
Install History	2014-02-07	2014-02-08	60730045
Plist artifacts	2014-02-13	2014-03-02	60934754

Table 3.1: Plaso review process.

Artifact	Plaso Plugin
Stored WiFi	airport.py
Apple account	appleaccount.py
Mac user accounts	macuser.py
Mac OS X updates	softwareupdate.py
Spotlight search terms	spotlight.py
Spotlight volume information	spotlight_volume.py
TimeMachine Backups	timemachine.py
Bluetooth devices	bt.py

Table 3.2: Property List Plaso implementation.

New parsers

With the new version comes a lot of new parsers, many of which contributed by various plaso developers. Special thanks to the tireless efforts of Joaquin Moreno for adding tremendous amount of Mac OS X support.

- Generic (OS independent):
  - Bencode parser [Brian Baskin]
  - Browser cookie parser
  - Chrome Cache files
  - CUPS IPP [Joaquin Moreno]
  - Firefox Cache files [Petter Bjelland]
  - Opera history files
  - XChat log [Francesco Picasso]
  - XChat Scrollback log file [Francesco Picasso]
- Mac OS X - all of which were contributed by Joaquin Moreno:
  - Basic Security Module (BSM)
  - Apple System Log (ASL)
  - Filewall log (appfirewall.log)
  - Keychain file
  - Securityd logs
  - Wifi log (wifi.log)
  - UTMPX
- Linux
  - PopContest [Francesco Picasso]
  - UTMP [Joaquin Moreno]

Figure 3.4: Plaso 1.1 released on 6th of June.

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.
Dennis Ritchie

4 Mac OS X

This chapter will explain how Mac OS X was design, the design reasons taken by Apple and the forensics implications of these design decisions.

4.1 Mac OS X Design

During the 1998 Worldwide Developers Conference Apple started to work in a new system. It was in March 1999 when Apple instead of release the DR3 Rhapsody announced the new operation system: Mac OS X Server 1.0. Apple launched four Developer Preview release (DP) of Mac OS X. In the first DP, the support for Mac OS Classic application (version 8) was implemented using the Carbon API interface called the Blue Box. In the second DP the support for Rhapsody applications (Mac OS 9) was implemented using the Cocoa API interface called Yellow box. During the San Francisco Macworld Expo in January 2000 the third DP was released implementing the Mac OS X GUI native interface called Aqua. In the last DP (fourth) the Dock and the Finder was implemented. It was in September 2000 when the Mac OS X public beta was release and finally in March 24 of 2001 the first Mac OS X 10.0 was released with the codename Cheetah [84]. Hence, Mac OS X has support for three difference GUI interfaces: Carbon (Classic or version 8), Cocoa (Rhapsody or version 9) and Aqua (Mac OS X, version 10).

Mac OS X was designed using three types of technologies: the originated by Apple, the originated by NeXT and the rest part where usually they are open source technology. A brief resume can be showed in the Figure 4.1. The kernel of Mac OS X is called **XNU** which is based on: I/O Kit, Mach and BSD [84]. The core of the kernel is based on Mach 3.0 wrote in C where the main goals were the hardware abstraction, memory management, scheduling, multitasking, virtual memory, low-level IPC and real time support. The device drivers framework are implemented using the object orientation I/O-Kit which is written in a embedded C++.

Over these two technologies, exists the BSD layer based on Free BSD 5.0 witch goals are provide signals (converting the exceptions from Mach), POSIX, System V,

Virtual File System, ACLs, TCP/IP stack, sockets, User ID and permission. Both, the BSD and Mach share the same memory directions working as a privileged level. Then, despite Mach is a microkernel, XNU is a monolithic kernel due that it is the combination between Mach and BSD [59]. Also, XNU permits to load modules directly without the requirement of recompile the kernel. These modules are called Kernel Extension [16].

The operation system is called **Darwin** which is the combination between the XNU kernel, basic libraries and a group of system utilities. Entirely Darwin is under Apple Public Source License, and then, the source code is public available. However, the source code from the upper layer: core services, Quartz, Carbon, JVM, Quick Time and so on are not available and they have a restricted Apple license.

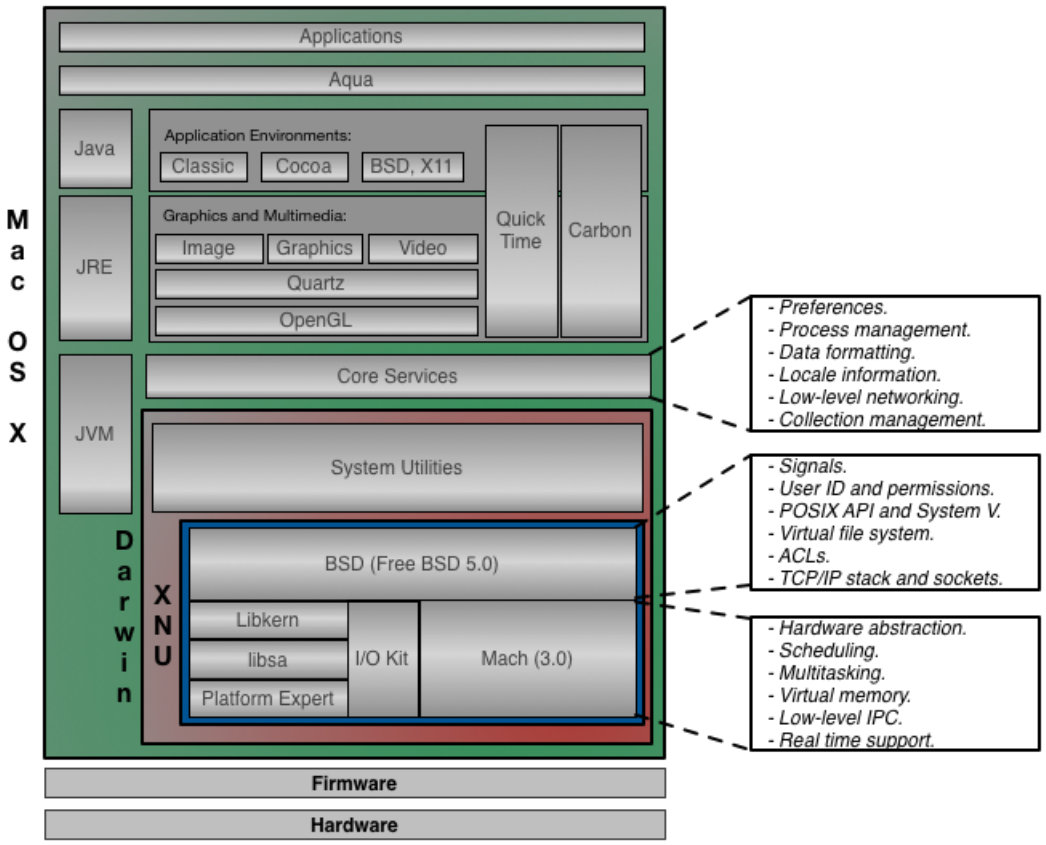

Figure 4.1: Mac OS X structure

4.2 Timestamp

Due to the design decision during the development of Mac OS X and the fact that it takes code from different sources and the dependencies with previous Apple implementation, the timestamp is represented using four different formats depending on the source of the data and the layer where the data comes from [76]:

- **HFS Time**: it is used in HFS file system and it is saved as a 4-byte hexadecimal value. The timestamp represents the number of seconds since January first of 1904 [68].

- **Epoch**: it represents the number of seconds since January first of 1970 [2]. It is also called *Posix* or *Darwin time*. It can be saved as an integer, float or hexadecimal value. If it is saved as an integer value can be follow by dot "." and another integer number that represents the microseconds of the timestamp. Some other applications store the timestamp as a hexadecimal value using a 4-byte hexadecimal value. The hexadecimal notation can be follow by another 4-byte hexadecimal value that represents the microseconds. The most common way to save the hexadecimal timestamp value is uses the big ending format where the first byte is the most significant value. Whereas other few evidences, as an example UTMPX, store the information using 4-byte hexadecimal in little endian instead of big endian.

  As an example, the date "Mon, 25 Nov 2013 15:47:38 +7702" it is represented with the integer value "1385394470" or with microseconds "1385394458.770200". In hexadecimal the value is represented as a "1a 71 93 52" in little endian or "52 93 71 1a" in big endian. If it has microsecond, it is represented adding another 4 bytes before the Epoch timestamp as a "98 C0 0B 00" in little endian or "00 0B C0 98" in big endian.

- **Cocoa**: it is used by some GUI applications and property list attributes. The timestamp represents the number of seconds since January first of 2001. It might store as an integer value or as an integer value follow by *.dot* and the microseconds. It is also called *CFAbsoluteTime* [4] or *NSTime* [15] depending on the documentation.

- **Plaintext**: some evidences store their timestamp using a clear plaintext where the information is human readable. The most common way to store their timestamps is using the BSD plaintext format *Month Day HH:MM:SS*. Other implementation can store the timestamp using the Mac OS X plaintext format *YYY-MM-DD HH:MM:SS.MSC(6)* and other specific formats such as *wifi.log*: *DayWeek Month Day HH:MM:SS.MCS(3)*.

  However some of these human readable evidences do not save the year making difficult to create a valid timestamp. For this reason, when these kinds of evidences are parsed, the creation file time (Ctime) is considered as the start year of the evidence. The parsers will store the year from Ctime as a generic variable, and also, the month from the previous line log. When a new line has a month earlier than the previous entry, it increases the year in one. As an example, if the month from the previous entry was November (11) and the new entry is February (2), it compares if February is less than November, and then, it increases the year. This approach works only if the difference between two adjacent entries has a timestamp difference less than eleven month. In case of Mac OS X, the bigger difference between two consecutively entries was less than 1 hour making this approach valid for this operation system.

Additionally, forensics investigators need to work with the same timestamp to be able to correlate the evidences in the same time line. Different solution should be taken to work with different timestamps. The first approach can be translating all the timestamp to the human readable date time; however, this approach has very bad impact performance. For this reason, instead of use human readable date time, the software engineering decide to use an integer value, and only when the information is represented, the timestamp is translated to human readable date time.

The approach implemented during the dissertation will use an integer timestamp. The next step to work with timestamp values is decided what integer timestamp representation should be used. Some engineers decided to uses the newer timestamp (Cocoa timestamp for Mac OS X) for performance issue due to the fact that the integer number is smaller: less difference between the initial time and the evidence time, whereas other investigators prefer to use the oldest one (HFS timestamp for Mac OS X) trying to avoid the time gaps when an evidence has a timestamp older than the smaller time that can be represented.

In our case, and due to the fact that Plaso is a multi operation system platform, the Epoch timestamp is going to be used as a base timestamp being compatible with other operation system. The timestamp must be synchronised with Epoch timestamp deleting *2082844800* seconds from HFS timestamp evidences and adding *978307200* seconds to Cocoa timestamp evidences (Figure 4.2). Taking into account that when the timestamp evidence is not correct, it does not exist or it is never been accessed the timestamp value is going to be the base time from its timestamp representation, or in other words, the timestamp 0. Then, if a timestamp with Cocoa format is from January First of 2001 or HFS Timestamp from January First of 1904 to December 31th of 1969, these timestamps must be changed to the Epoch base time or *0 second*: *January First of 1970*.

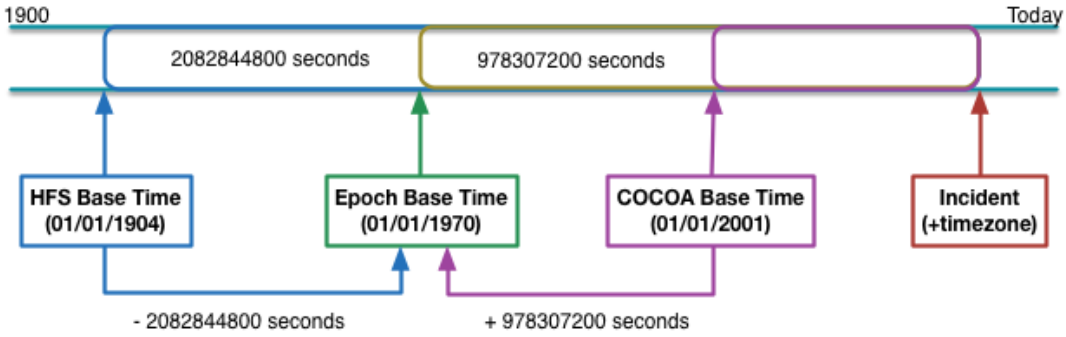

Figure 4.2: Mac OS X Timestamp

Using as a time base the Epoch timestamp, all timestamps need to be represented using the same format representation, where two approaches can be used: float integer or a long integer. If it is represented as an float integer, the integer part represents the seconds and the decimal part the microseconds. If it is represented as long integer, the timestamp is represented in microseconds where the timestamp is calculate as a: "seconds * 100.000 + microseconds". The second one has a performance benefits due

to the fact that long integer number is faster than floating point numbers, but it is less clear than the first approach.

Additionally, a few application evidences provided the timestamp based on the Mac OS X timezone. Then, if Mac OS X was configured with the Easter Coast of USA, all the timestamps are going to have five hours less than the GMT+0 time. The timestamp must be normalized to the same timezone, usually UTC.

4.3 Hierarchical File System Plus (HFS+)

Hierarchical File System Plus (HFS+) was developed by Apple to substitute the HFS as a default Macintosh file system in 2000 [27]. HFS+ added some improvement over HFS supporting 32-bit length for block address and allocation mapping table, encoding the file and directory names using Unicode (UTF-16) and adding the concept of n-forked [11]. Furthermore, In Mac OS X 10.4 Apple added the Access Control List (ACL) functionality to the HFS+ file system.

HFS+ has different elements such as volume header, catalog file, extend overflow, allocation file and startup file. Focusing on forensics purpose, the most important element is the Catalog File, a B-tree structure that contains the metadata of all files and folders: times, user and group (HFSPlusBSDInfo), permission, file id, etc [27]. Regarding to the timestamp, it contains four different timestamp: the last time when the metadata of the file was modified (modified time o crime), the last time when the file was acceded (atime or access time), the last time when the data of the file was modified (mtime or modified time) and finally the time when the file was created (crtime or created time). All the timestamp fields are an unsigned 32-bit integers (UInt32) value with the seconds since midnight, January first from 1904 in GMT time [49]. Then, as an unsigned 32-bit integer, it can store timestamps until February 6th of 2040 at 06:28:15 GMT. Also, the developers must be aware to convert the GMT value into local time during the application implementation.

Each file and directory in the Catalog File contains attributes called DataFork and ResourceFork. Both of them are HFSPlusForkData type witch contains two important attributes: logicalSize and totalBlocks. The logicalSize contains the bytes of the data assigned or reserved to the file (indeed the fork) whereas the totalBlocks contains the number of blocks allocated, used, by the file. Regarding to DataFork, it contains the location and the size of the reserve file to this file whereas ResourceFork contains relevant data and resource information. Moreover, each file in the catalog contains a HFSPlusExtentDescriptor structure that indicates the number of the first block (startBlock) and the number of blocks in that position (blockCount) [27].

In other words, the Catalog File provided the list of files and directories of the partition. For each file and directory, the catalog provides the name of the file, the timestamps and using the combination of DataFork and the HFSPlusExtentDescriptor attributes the information contained by the file can be extracted.

Finally, HFS+ file system has an element called Volume Header that contains all the

information related with the volume that contains useful metadata. The most important is the timestamp when the volume was created. This timestamp is stored in local time and not in GMT because this timestamp is used as a relative unique identified [49].

4.4 Bundle Layout

Bundle is a hierarchy directory that contains different files and directories than it is used as a resource package [84]. The root directory is showed by the GUI interface (Finder) as a only one file whereas the console interface showed it as a common directory. The bundle is used to store executables, libraries, documentation, etc. The bundle can be configured using a property list file (Info.plist), PkgInfo file or using a file system attribute called kHasBundle.

The most common bundle is the application bundle [67] used to store the application as a self-container in a manner than it is quite easy to manage applications and their dependencies making trivial to the final user the process of installing and removing applications. Usually this application bundle has the ".app" suffix. It provides an abstraction layer where different version of shared libraries and binary may be stored inside the application bundle to execute this application in different version of Mac OS X and in different architecture such as Power PC, X86 or X86_64.

Some malware uses the bundle layout to hide files and change the path of execution of the common program in a manner than when the program is executed by the user, instead of execute the expected program, the malware is executed in first instance, and after that, the expected program is executed by the malware.

4.5 File Vault

In Mac OS X 10.3, Apple developed a functionality called File Vault to provide certain level of confidentiality where the home directory of the user was encrypted using AES 128 [27]. This protection has the goal to prevent information leak of the owner if the computer is lost or stolen. Some version after, in Mac OS X 10.7, Apple developed a volume encryption file system called File Vault 2 that encrypt a whole partition using the AES-XTX encryption algorithm where the entirely partition is encrypted and not only a directory [13].

Apple has not release the specification related to FileVault2 making almost impossible the capacity to analyse an evidence without altering it. The only option for the incident response team was to mount the partition in read only as a second hard disk using another Mac OS X system as a host device. Nevertheless a group of investigators release a paper [65] and a library called Libfvde [64] that explains how the encryption mechanism works and how an incident response investigator is able to decrypt these partitions without altered the evidence and without require a Mac OS X environment as a host.

The decrypted partition can be recovered using two different mechanisms: the de-

cryption password or the recovery key. Libfvde provides support for both of the two methods. Both of them requires extract the *EncryptedRoot.plist.wipekey* property list from the Recovery partition, using as an example The Sleuth Kit software. In figure 4.3 an example of decrypting and mounting a FileVault2 Partition using the library Libfvde is showed where the chose mechanism was the password method.

```
root@plaso:/images# mmls filevault2_10.9.dd | grep "Recovery\|Mac"
05:  01      0000409640   0082616503   0082206864   Macintosh HD
06:  02      0082616504   0083886039   0001269536   Recovery HD
root@plaso:/images# fls -r -o 82616504 filevault2_10.9.dd | grep EncryptedRoot
+++++ r/r 180:  EncryptedRoot.plist.wipekey
root@plaso:/images# icat -o 82616504 filevault2_10.9.dd 180 >> ER
root@plaso:/images# xxd -l 10 ER
0000000: 3343 8e0c 907a 2db3 ccb4                 3C...z-...
root@plaso:/images# OF=$((512*409640))
root@plaso:/images# fvdemount -e ER -p abcd -o $OF filevault2_10.9.dd mount/
fvdemount 20130305

root@plaso:/images# mkdir mount2
root@plaso:/images# OPT='ro,noexec,loop,umask=0222'
root@plaso:/images# mount -t hfsplus -o $OPT mount/fvde1 mount2/
root@plaso:/images# ls -l mount2/mach_kernel
-rwxr-xr-x 1 root root 8393256 Sep 20 06:22 mount2/mach_kernel
root@plaso:/images#
```

Figure 4.3: Decrypting a partition using Libfvde library

4.6 Evidence Types

Mac OS X stores a group of important forensics evidences inside the file system. These evidences are stored using different types of formats. Some of them are well known such as ASCII plain text, XML or database files; whereas others are binary structures used only by Mac OS X.

Binary files have a few or do not have the required documentation to understand how the information is stored. For this reason, a reversing engineering is required to try to understand how this information is saved. The first and easy approach to understand the format files is read the content of the file using an hexadecimal editor that can represent the information as a hexadecimal values, binary values or ASCII character representation.

The approach done to analyse these evidences was applied in different steps. Some binary files have header and one or more entries, whereas other do not have header. The identification if the binary has header, when the header ends and when the entry starts must be done to identify which part of the file corresponds the header and witch part the entry. Having this information, an analyse process must be done to try to recognise the information taking into account ASCII strings and the integer value from

the hexadecimal combination between: nibble, byte, 2 bytes, 4 bytes and 8 bytes that can be saved in Little Endian or Big Endian that can represent values such as timestamp, length, UID, GID, PID, IP address, return values, etc.

Apple usually provides its own tools than can read these evidences. Hence, having the output from this tools and having the binary files as an input, the researcher can try to match the values to understand how the storage process works. In this subchapter, a summarise of the different file types that they will be explained in future chapters:

- **Basic Security Module** (BSM): binary files that store the kernel audit logs. For each BSM file it has one or more entries. Each entry is a group of C structures called tokens where each of these tokens saves an specific type of information. Some of these tokens has a static size whereas other have different sizes.
- **ASL** (Binary Apple System Log): it is the default daemon log service for Mac OS X called Apple System Log. By default ASL stores the information using the binary structures NSLog using a doubly linked list of entries.
- **Keychain**: binary database files that contains tables to store different type of records. The file is used to store passwords and the certifications from applications, Internet resource, networks such as wifi or bluetooth, and so on.
- **Binary own format**: some evidences store their information using their own binary structure of data. As an example UTMPX, Cups control files, Java IDX and Binary Cookies.
- **Plist**: files used by Mac OS X and external applications. It is used to store application and system configurations. They can be stored in two different formats: XML and Plist binary format called BPlist.
- **Plain text**: files than required a regular expressions to be parsed. The timestamp is saved using a formatted time (not numeric value) where usually the year is omitted. These files represent a timestamp and performance challenge.
- **XML**: well know files with an specific structure. A common XML parse library needs to be used to parse this structure. The auditor needs to identify which structures and how they are saved, and then, if the structures can provide useful information for the case.
- **SQLite**: small database that it is used because is easy to get access and save the data without need a heavy data base server. It can be parser by sqlite3 client, programming library or a GUI interface, as an example Sqliteman. Usually contains the log from user applications such as Skype, Chrome or Firefox and some caches from Mac OS X.

These evidences are stored in the Mac OS X file system using one of the file types explained above. Each of them are located in figure 4.4 and they will be explained in future chapters.

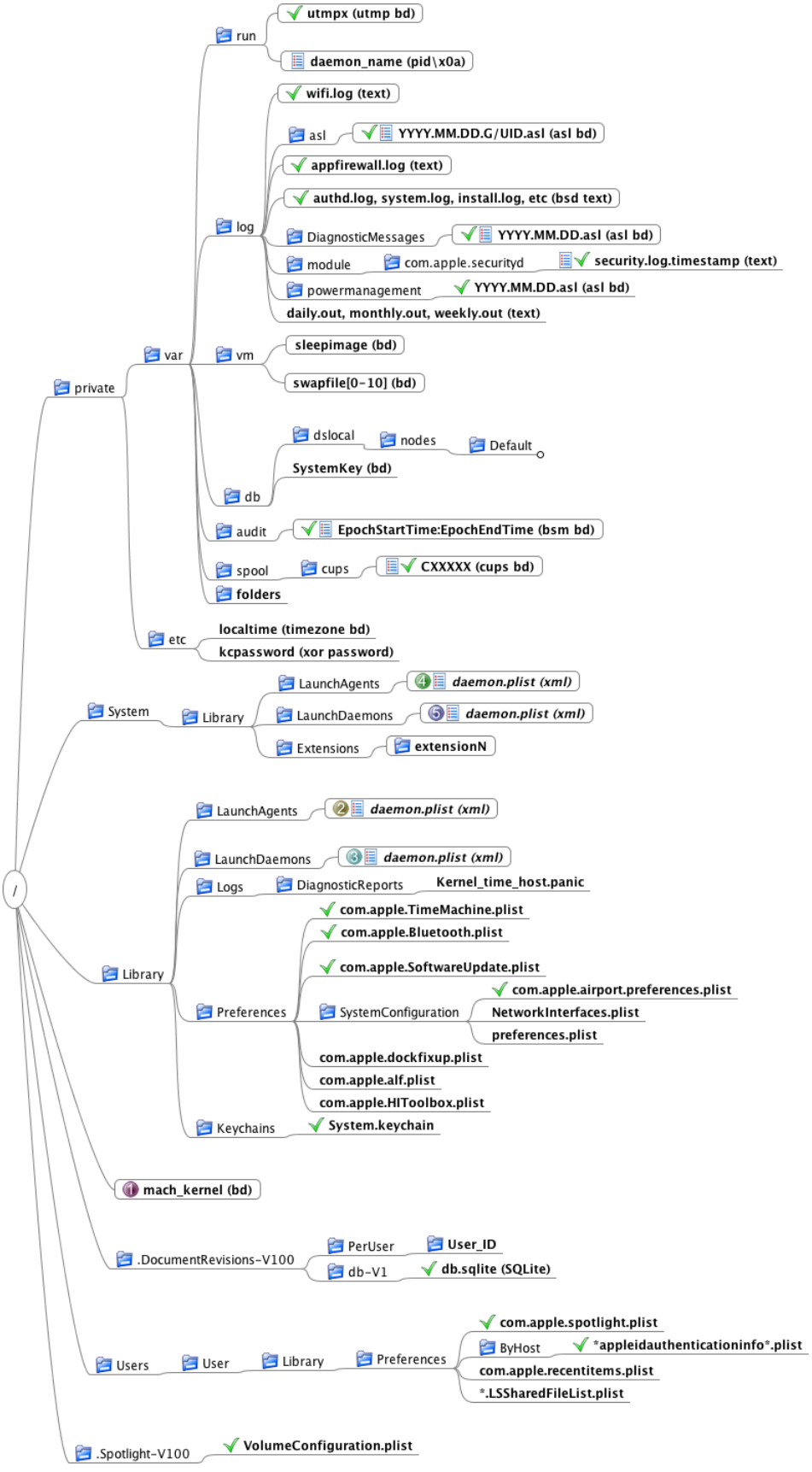

Figure 4.4: Mac OS X Evidences.

A good programmer is someone who always looks both ways before crossing a one-way street.

Doug Linder

5 Mac OS X No Time Valuable

During the first step of file system analysis, the auditors should extract some not timetable valuable that provided useful information to correlate adequately the timestamp evidences. Moreover, these preliminary evidences can provide some clues to the forensic analyzers that can resolve the incident. As an example, the list of process that are executed during the system boot have not a timestamp valuable, but they might provided useful information if one of this process is not a well know process identifying a malware behaviour.

5.1 System basic configuration

The system basic configuration is a group of files, usually binary property lists that have the basic information required to parse adequately the timestamp evidences.

All the extracted timestamp evidences must be correlated using the same time base. Having all the evidences with the same timestamp they can be ordered using the timestamp as a reference, creating the super timeline where all the extracted evidences are reflected. Some evidences use the local time of the computer to store the time, whereas other uses the base timestamp or GMT+0. The auditors and the tools need to identify the evidences that use the local time and translate their timestamp to GMT+0 in a manner than all the evidences use the same base timestamp. In Mac OS X, the file that contains the local time of the system is in the file *localtime*:

```
/private/etc/localtime
```

Another important prerequisite information is the name of the computer that must be known. Some evidences provide the host name or computer name as a part of the evidence information, whereas others provided the network address (IP) of the host. The name of the host is provided in the binary plist *preferences.plist*:

```
"/Library/Preferences/SystemConfiguration/preferences.plist"
BPlist Token: "System -> System -> ComputerName"
```

Due to some characters and encoding issues, it is important to know the keyboard layout avoiding future parsing problems:

```
/Library/Preferences/com.apple.HIToolbox.plist
BPlist Token: AppleCurrentKeyboardLayoutInputSourceID
```

During the evidence acquisition, the auditor can decide to store the network traffic using a TAP or port mirroring, and also, the networks logs obtained from the network devices, such as intrusion detection system or switch. For this reason, known the network interfaces and their MACs can be useful. Mac OS X stored this information using a binary plist file *NetworkInterfaces.plist*:

```
/Library/Preferences/SystemConfiguration/NetworkInterfaces.plist
BPlist Token: Interfaces (for each item) -> "BSD Name" contains the
    interface name and "IOMACAddress" the MAC address.
```

Mac OS X has an application firewall as a network security mechanism. Knowing if the firewall is activated, and if it is, the firewall rules can reveal some malicious behaviour. The application firewall is configured using another binary plist file *om.apple.alf.plist*. If the attribute *global state* has the value *1* the firewall is activate or *0* if not. The file has two list of attributes, the *exceptions list* that identify when the traffic is always permitted, using the path field as identification, and the application list that contains the list of the applications and the rules applied to each application:

```
/Library/Preferences/com.apple.alf.plist
BPlist Token:
exceptions (for each item) -> path
applications (for each item) -> "bundleid" the application name (no
    mandatory) and the "state": if 0 the traffic is accepted or 2 if the
    traffic is dropped.
```

Finally, Mac OS X stored all the printer jobs using the CUPS printing system. To help the CUPS parser in the timestamp valuable evidences chapter, the parser should extract all the available printers and if the printers are activated to print the jobs:

```
/Library/Preferences/org.cups.printers.plist
For each item -> Printer-name attribute contains the printer name whereas
    the conditional printer-is-accepting-jobs attribute is true when the
    printer is activated.
```

5.2 User accounts

During a forensics analyse it is important to know the actions related with the system users accounts and their identification ID related. Some evidences use this ID instead of the user name where the pair UID and user name should be correlated as the same entity. In Mac OS X, the system user information is stored using a binary plist file:

```
"/private/var/db/dslocal/nodes/Default/users/username.plist"
```

This Binary Plist file contains useful attributes that can be extracted using the tool *mac_password.py* provided in the Appendix A (Figure 5.1) or a *macuser* plist plugin in Plaso:

- shell: the shell path used by the user.
- realname: the first and last name of the user.
- name: system user name.
- home: home directory.
- uid: numeric user ID that identify the user in the system.
- gid: numeric group ID of the user.
- passwordpolicyoptions: a XML Plist with timestamp attributes such as the last time that the password was change, the number of incorrect attempts, etc. This field is explained in the next chapter.
- ShadowHashData: a Binary Plist with the information required to authenticate the user. Since Mac OS X 10.8, the password is stored using the PBKDF2-SHA-512 [53] derivation function algorithm. This attribute contains the required three attributes to validate the input password:
  - Iterations: number of times that the action is repeated. Usually around 37000.
  - Salt: provided the capacity to have different hashes despite the passwords is the same. Providing some protections against rainbow tables.
  - Entropy: a hash with 512 hexadecimal characters resulting of the PBKDF2 operation between the real password, the salt value and the number of iterations.

Apple uses the PBKDF2-SHA-512 algorithm since the last two versions of Mac OS X which is hard to be recovered due to the performance impact of the iterations required to check if it is a valid password. Using the tool Dave [94], a performance around 10-15 word per second (Figure 5.2) is obtained, whereas Hashcat [90] checks between 25 and

```
DarkTemplar:Password moxilo$ python mac_password.py moxilo.plist

User: moxilo
UID: 501
GID: 20
Shell: /bin/bash
Policy:
  failedLoginTimestamp at 2001-01-01T00:00:00Z.
  lastLoginTimestamp at 2001-01-01T00:00:00Z.
  passwordLastSetTime at 2013-12-28T04:35:47Z.
Available Passwords:
 Mac OS X user password:
  Iterations: 37313
  Salt: fa6cac1869263baa85cffc5e77a3d4ee164b75536cae26ce8547108f60e3f554
  Entropy: a731dbb0e386b169af89fbb33c255ceafc083c6bc5194853f72f11c550c42e4625ef1
13b66f3f8b51fc3cd39106bad5067db3f7f1491758ffe0d819a1b0aba20646fd61345d98c0c9a411
bfd1144dd4b3c40ec0f148b66d5b9ab014449f9b2e103928ef21db6e25b536a60ff17a84e985be3a
a7ba3a4c16b34e0d1d2066ae178
 Kerberos:
  Version: Kerberosv5
  Hash: LKDC:SHA1.071E456555B80E2C4DC25BE20CEFA634B3882461
```

Figure 5.1: User information extracted from mac_password.py.

30 words per second. For a scenery of a real brute force attack against 5 character password with uppercase, lowercase and numbers the average case required 5 month to recovery the password.

During the analyse, the author found a implementation error in Hashcat. If the tool is used over a x86 architecture using the mode 7100 (PBKDF2-SHA-512 algorithm). During those conditions, the tool was not able to decrypt the password. The error is reported to Hashcat develop team and solved in the version 0.48.

```
bash-3.2# ./dave -p moxilo.plist -j moxilo -d wordlists/own
moxilo:$ml$37313$fa6cac1869263baa85cffc5e77a3d4ee164b75536cae26ce8547108f60e3f554$
a731dbb0e386b169af89fbb33c255ceafc083c6bc5194853f72f11c550c42e4625ef113b66f3f8b51f
c3cd39106bad5067db3f7f1491758ffe0d819a1b0aba20646fd61345d98c0c9a411bfd1144dd4b3c40
ec0f148b66d5b9ab014449f9b2e103928ef21db6e25b536a60ff17a84e985be3aa7ba3a4c16b34e0d1
d2066ae178
-- Loaded PBKDF2 (Salted SHA512) hash...
-- Starting attack

-- Found password : 'abcd'
-- (dictionary attack)

Finished in 0.773 seconds / 11 guesses...
14 guesses per second.
bash-3.2#
```

Figure 5.2: Dave tool recovering a Mac OS X 10.9 password.

5.3 Autologin

Mac OS X permits automatically logon on into the system without require that the user provided the password. The binary plist file *com.apple.loginwindow.plist* indicates if the user can automatically logon in the environment:

```
/Library/Preferences/com.apple.loginwindow.plist
"autoLoginUser" contains the name of the user that can automatically
    logon on the system.
```

To permit this action, Mac OS X needs to store the password in the way that can be recovered, and then, automatically logon the user in the system without require the password. However, to permit this functionality the password cannot be encrypted. Mac OS X stores the password in the *kcpassword* file using a XOR operation against the user password before stores it. Each password character is stored using a 1-byte representation (ASCII numerical code). The XOR operation is against a magic 11 bytes value *0x7d 89 52 23 d2 bc dd ea a3 b9 1f* [89] where the first byte of the password (first character) is xored against the first byte of the magic number, the second of the password against the second of the magic number, and so on. If the password has more than 11 characters the magic number is repeated.

Before this process, Mac OS X pads the end of the user password with 0x00 to have a multiple of 11 bytes being able to recognise when the password finished. In the appendix A, a tool called *kcpass.py* has been implement to extract the password providing as an input the kcpassword hexadecimal value:

```
# xxd /etc/kcpassword
0000000: 1ceb 3147 d217 2f11 40ff 63bf        ..1G../.@.c.
# python kcpass.py 1ceb3147d2172f1140ff63bf

       Kcpasswd: 0x1ceb3147d2172f1140ff63bf.
       Magic Xor: 0x7d895223d2bcddeaa3b91f.
       The password is: "abcd".

#
```

5.4 Recent Files

In Mac OS X 10.5, Apple provided a new Launch Services Framework [6] that contains the list of the last opened files. The API permits to load and share file lists using the LSSharedFileList library. It is common used by different tools where for each user, it has in its *Library/Preferences* directory a list of Plist binary files for each application that uses this API and a specific file called *com.apple.recentitems.plist* for the last files opened by Mac OS X [33].

```
/Users/user_name/Library/Preferences/
       com.apple.recentitems.plist
       companyA.applicationA.LSSharedFileList.plist
       companyB.applicationB.LSSharedFileList.plist
```

```
...
companyN.applicationN.LSSharedFileList.plist
```

The Plist files contains a list of item called *CustomListItems* where for each item it provides the *Name* attribute that contains the filename of the opened file and a *Bookmark* attribute. The *Bookmark* attribute is a unknown binary data created by the LSSharedFileList library witch source code is not available (only the headers). After doing a reversing process, different field have been identified: the string file path, the inode file path, the name, UUID and mount point of the partition where the file is stored, some sandbox attributes. Also, if the file was opened from an external device, it contains the path to this external device. Additionally, the string file path does not change if the file is moved whereas the inode file path is modified to the new file path, assuming that the library follow the inode path instead of the file path. The resulting reversing process of the bookmark file can be observed in the figure 5.4.

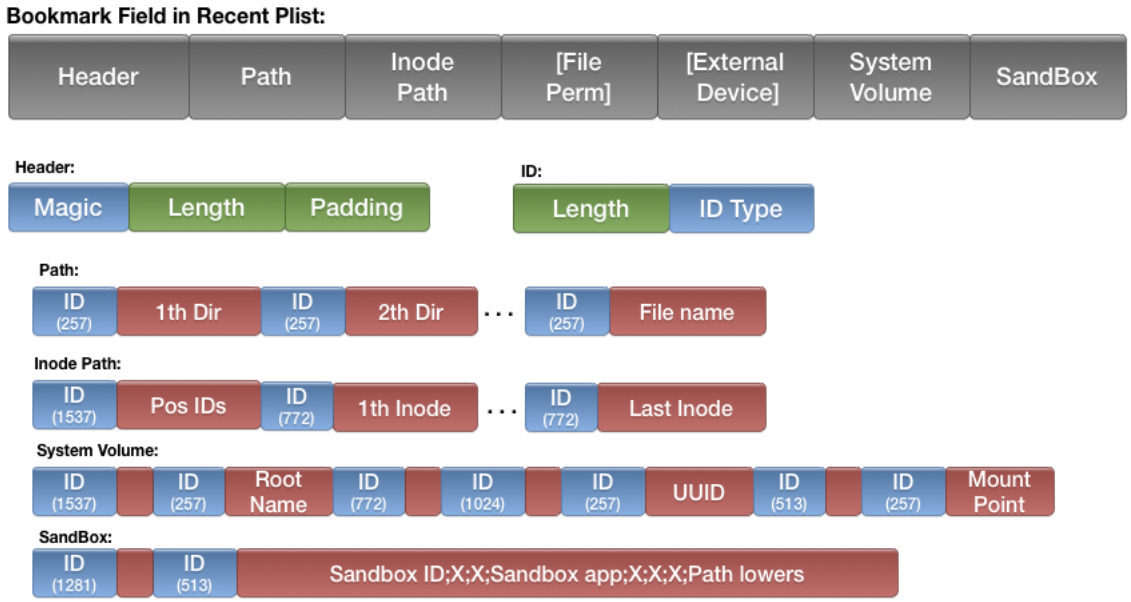

Figure 5.3: Bookmark Plist attribute.

The bookmark object is stored using little endian approach. The object is divided in different attributes with a specific format. An attribute is a 4-byte field that represents the length of the attribute follows by 4-byte field that indicates the type of the attribute, and finally, the information contained by the attribute. The information and its format depend on the attribute. A specific attribute is the attribute that contains other attributes represented by *0x0106*. This attribute list is follow by 4-bytes field that contains the offset where the attribute starts. The attribute length only counts the information part without the 4-bytes used for indicate the length and also the 4-byte used to indicate the type of attribute. Finally, all the strings values are padding with *0x00* character being a 4-byte multiple. An hexadecimal representation can be observed in figure 5.4.

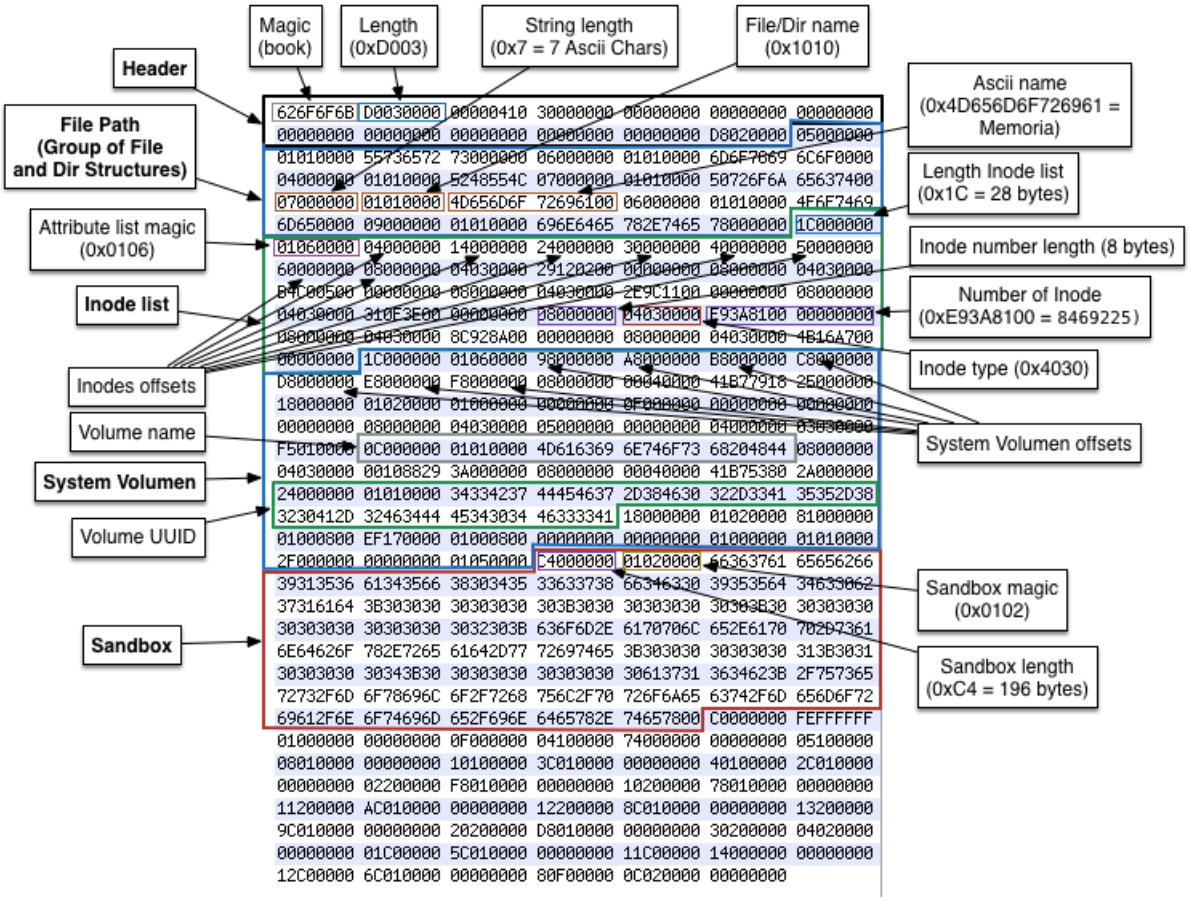

Figure 5.4: Bookmark hexadecimal Plist attribute.

The tool *mac_recent.py* has been implemented as a proof of concept of a tool that can extract the bookmark information witch source code is available in *Appendix A*:

```
$ python mac_recent.py com.apple.TextEdit.LSSharedFileList.plist
File: com.apple.TextEdit.LSSharedFileList.plist

       Recent document open by TextEdit(apple): plist.txt
       Path: /Users/moxilo/RHUL/Project/Parsers/Plist/plist.txt
       Inode Path:
           /135721/377012/1154094/4066865/8026468/10881261/10881752
       HD Partition Root Name: Macintosh HD
       HD Root UUID: 43B7DEF7-8F02-3A55-820A-AAAABBBBCCCC
       HD Root mount in: /
       Sandbox ID: f28f331fd40e4c5f690043be700eba0e90000000
       Sandbox Path: /users/moxilo/rhul/project/parsers/plist/plist.txt
...
```

5.5 Kernel, Extensions, Swap and Hibernation File

Mac OS X stores and uses some valuable binary artifacts. The first artifact is the XNU kernel itself that can be found in the root directory with the name of *mach_kernel*, taking into account that as it was describe before it is not a mach kernel, it is XNU kernel: *I/O Kit + BSD + Mach*. A hash function against this file should be applied to know if the kernel has been modified.

```
/mach_kernel
```

XNU permits to load a kernel extension, called drivers or modules in other operation systems, to be allowed to extend the kernel functionalities without need to be compiled inside it [58] . This kernel extensions are stored in the directory */Library/Extensions/* where for each extension it use a *application bundle layout* with the name *name_extension.kext*. This directory must have the *Contents* directory where inside this directory it has the files *Info.plist*, *version.plist* and the directory *MacOS*. These two files provided the information related to the kernel extension and the binary module itself is inside of this *MacOS* directory. Mac OS X provides the tool *kextstat* to know the status from the different extension. Also, the extensions can be lunched or removed with the *kextload* or *kextunload* tools.

The last version of Mac OS X stored the swap and hibernation file in the */private/var/run* directory [24] where the swap file can be stored in a maximum of eleven files (from 0 to 10). Nowadays the forensics and reversing malware engineers only can use these files to extract some useful ASCII character values that can provided some clues about the incident. It has different reason that make difficult work with swap files [38]:

1. The raw memory pages stored in the swap files are not organised.
2. The information stored in the swap files might not clean after reboot a system containing information from a previous boot.
3. Allocated disk blocks are not sanitise in some operation systems.
4. Since Mac OS X 10.7, the swap files are encrypted using 128-bit AES keys.

```
Hybernation file: "/private/var/run/sleepimage"
Swap file: "/private/var/run/swapfile[0-10]"
```

Finally, since Mac OS X 10.9 the operation system has used a modern feature called compressed RAM, where the information in the memory is compressed trying to avoid swapping the information due to the performance reason. This new manner to work with the memory required a parser to translate the acquired encrypted memory to a non-encrypted memory image [38].

5.6 Booting and persistence

Operation systems have an ordered group of steps to initialise the system with the required applications. During this process, the operation system uses different interfaces to execute a set of process that provided different services: drivers to support some hardware, requires network services, programs, etc. This boot process can be used by the attackers to install a malware in a manner that if the system is rebooted the malware will be executed during the next boot process. This functionality is called *Persistence*.

Knowing where the software can be executed during a boot process is an important source of information to detect abnormal executable files. Mac OS X boot process is divided in 5 main steps: bootrom, boot.efi, XNU, launchd and finally loginwindow [95, 84, 58]:

1. **Bootrom**: is the combination between POST (Power-On Self Test) and EFI (Extensible Firmware Interface). EFI is an interface with boot loader programs, where Apple EFI has its own implementation. It differs from the original Intel initiative 1.1, providing more functionalities and being closer to Universal EFI approach. The purpose of *bootrom* is verify the memory, initialise the hardware and finally selecting the boot partition. Despite *bootrom* might not be considered as a target, the predating A5 chips had a vulnerability in its bottom where an attacker was able to exploit with *limera1n exploit*. EFI uses the *boot.efi* binary to load the kernel.
2. **Boot.efi**: the file is stored in */System/CoreServices/boot.efi*. The boot.efi is a binary with a fat header instead of being a PE header like other EFI implementations. It contains fields such as the signature, number of architectures, the type

of processor, the architecture type, the offset of the executable, etc. The goal of *boot.efi* is initialise the device tree, locate the kernel, load the boot kexts and jump to the kernel using the EFI RunTimeServices and initializeConsole to call the NVRAM variables. Hence, boot.efi is the responsible to call the kernel XNU.

3. **XNU**: it is the kernel of Mac OS X stored in the directory */mach_kernel*. When the kernel is executed it initialise the match components, then the IOKIT responsible to provide the hardware drivers and load the kernel extensions, and finally running the BSD Subsystem. The BSD subsystem is the responsible to execute the first process and father of the next process: *launchd*. It is the Init process in other Unix and Linux environments.

4. **Launchd**: first process of the system and responsible to start the environment executing the required process. It is stored in the directory */sbin/launchd)*. The main goal during the startup process is run all the daemons and agents, and finally, executing the *loginwindows process*.

5. **LoginWindow**: GUI interface responsible of authenticate users, set up environments and manage the sessions.

Patrick Wardle has written an excellent presentation explaining how the Mac OS X malware manage the persistence, providing different malware approach. As an example, *CallMe* is execute as a daemon, *Flashback or Crisis* are executed as an agent, *Janicab is executed as a schedule task with crontab*, *Kitmos* uses the login item persistence. A resume of the most important persistence location is as follow [95]:

- Extensions: using a XNU extensions modules in the follow directories:

```
/Library/Extensions/
/System/Library/Extensions/
```

- Daemons: are a background program executed by the system. It is not related with any user and it is not allowed to connect to the graphic interface (windows server) [7]. They are configured using a XML property list. The directories where these daemons are installed are as follow:

```
/System/Library/LaunchDaemons
/Library/LaunchDaemons
```

- Agents: agents are a background program that is execute by a particular user. They are able to connect to the windows server. They are configured using a XML property list. The directories where these agents are installed are as follow [7]:

```
/System/Library/LaunchAgents
```

```
/Library/LaunchAgents
$home/Library/LaunchAgents
```

- Crontab: schedule daemon to execute task every period of time. It can be used to program a task that executes the malware every X time. The directories with the programmed tasks are as follow [5]:

```
/etc/crontab
/usr/lib/cron/tabs
```

- Login items: this is the official manner that a user can execute programs during startup. It can be store the data in base 64 encode. For each user, it has in her or his directory *$home/Library/Preferences/* a property list that called *com.apple.loginitems.plist* that contains the list of login items.
- Login / Logout hooks: the malware can be added to the login plist file *com.apple.loginwindow.plist* in the directory */private/var/root/Library/Preferences/* to be executed during the startup. It is a deprecated functionality that it is still supported by Mac OS X.
- Startup items: it is another deprecated functionality that permits lunch a binary during boot time. The directories with the programmed tasks are as follow:

```
/System/Library/StartupItems/
/Library/StartupItems/
```

- RC.Common: the init process (launchd) execute the rc.boot and the *rc shell scripts* to start the script. Also, it executed all the binary files indicates in the file */etc/rc.common*. Hence, a malware that add its binary to this files is able to be executed during next boot process.
- Launchd configuration file: the Launchd process has a functionality that permits to specify the execution of extra binaries. The path of the binaries must be added to the file */etc/launchd.conf*, not created by default. Hence, Launchd will execute during boot time the binaries added to that file .

My favorite things in life don't cost any money.
It's really clear that the most precious resource we all have is time.

Steve Jobs

6 Mac OS X Timestamp Evidences

The next chapter represents the dissertation core where the most relevant Mac OS X file system evidences that provided a timestamp value have been identified, analysed and explained.

6.1 Apple System Log (ASL)

Apple System Log (ASL) is a daemon that has the responsibility to manage and store the log information provided by the applications. The daemon is executed during the boot time from the configuration file */System/Library/LaunchDaemons/com.apple.syslogd.plist* that execute the binary */usr/sbin/syslogd* [58]. The tool *aslmanager* is the responsible to manage and rotate the logs generated by ASL. The information is stored using different criteria such as the application name that sends the information, the facility, the process identification, etc. Also, this information can be correlated with the priority level of the information. This level has eight numeric values from zero to seven, where as low as is the level the message has more priority. The text associate with each numerical level are defined in table 6.1.

Text level	Numerical level
Emergency	0
Alert	1
Critical	2
Error	3
Warning	4
Notice	5
Info	6
Debug	7

Table 6.1: ASL priority levels.

ASL was the substitute of the traditional Unix Syslog in Mac OS X 10.4. In the first version, ASL only stores the information using the BSD plaintext format in the file */private/var/log/asl.log*. It was in Mac OS X 10.5 when the binary ASL format was introduced, also known as a *store format*. Since this version the binary ASL format was used to save the majority of the information. Between 10.5 and 10.5.5 the information was stored in */private/var/log/asl.db*. Since 10.5.6 to 10.9.5 (last version) the files have stored in the directory */private/var/log/asl/* [81]. Inside this directory, ASL stores one file for each day from the last week using the format *YYYY.MM.DD.[UID].[GID].asl* for each file where UID is the numerical number associated with the system user and GID is the group identification number associate with the system group. These UID and GID identify the user and group that are able to read these files, plus the root user.

The daemon is configured by default by the file */etc/asl.conf*. Also, some specific applications are configured in the directory */etc/asl/*. The evidences are stored using two difference formats: binary ASL format and BSD plaintext format. In the following subchapter both of them are explained in deep to be allowed to extract the required information.

6.1.1 Apple System Log Binary File

Most of the applications use ASL to send their logs, and since Mac OS 10.5, the majority of these logs are stored using the ASL binary format. Despite the importance of these ASL binary files, the Apple's documentation only focuses on its own libraries (API) without explain how ASL works [12]. Moreover, only an ASL tool [26] has been developed (2012); however this tool is focused on some specific type of logs instead of focus on how ASL format binary file is stored.

For this reason, a binary reversed process must be done using hexadecimal readers combine with some Apple tools. As it was described in the process of understand binary files, the first step is identify if the ASL files has header and trying to know the size of the header and the meaning of the majority of the hexadecimal values stored in that header. Some times the header has some required data to be able to extract the entry. Knowing how the header works, the same process must be done for each entry. To help with this tedious process, the binary tool provided by Apple *syslog* is being used to understand how the binary data is represented in clear text. The *syslog* tool can be used as it is described below:

```
$ syslog -f file.asl -T utc -F raw
```

ASL file has one header and one or more entries. The header has a field that indicates when the first entry was created. Furthermore, it has two important fields: the first one indicates the file offset in bytes for the first entry and the second one the file offset in bytes for the last entry, providing the start entry and the last entry. Then, each entry is stored consecutively in the file where each entry has two pointers: a pointer to the previous entry and another pointer to the next entry.

Hence, ASL binary format is stored as a doubly linked list of entries where the tools that required reading this format can get easily the last entries, and at the same tame, the older entries (the first in the file). Taking into account than the next entry pointer from the last entry and the previous pointer entry from the first entry points both of them to the header or position *0x00*. In can be represented as the same manner than a list of process is stored in the majority of operation systems. An example of this design is explained in figure 6.1.

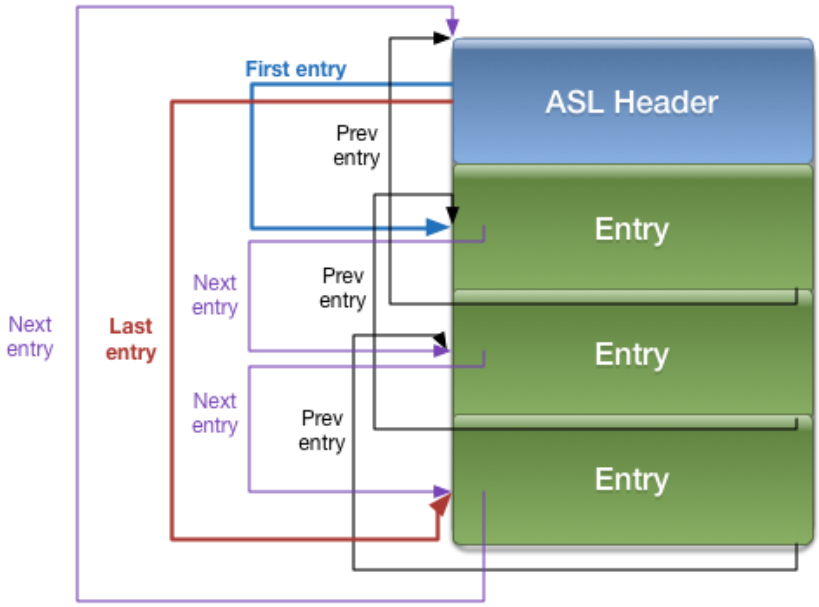

Figure 6.1: Generic ASL File structure.

Each entry is divided in three sectors. The first sector is the entry's heap where the dynamic information is stored, the second part is the structure of the entry where the static values are saved, and finally the end part of the entry created by 32 bytes (four 8-byte fields) plus multiples of 16 bytes (two 8-byte fields) for each extra fields in the entry and at the end a 8-byte field that has the pointer to the previous entry. Each entry might have different size because only the second part has a fix size.

In the static part of the entry, the first field has the size of the entry between the next byte until the end of the entry. Hence, the field's length itself and the heap is omitted in this value. The second field in the static part has a 8-byte pointer to the next entry follow by other fields such as the timestamp, nanoseconds, ASL message identification, level of syslog, the process PID / UID / GID that generates the entry.

The end part provided always the host, sender, facility and message, always in this order, that for each field it uses a 8-byte field. Furthermore, it can include other extra fields depending of the entry, where two 8-byte fields are used for each extra field: the first 8-byte saves the name of the extra field and the second saves the value itself. The field can contain two different representations: a string value if the string size is equal or lower than 6 characters or a pointer if it is bigger. If it is a String value with 6

or less characters, the information is stored directly in the 8-byte field where the most significantly bit is *1*, and then, the first nibble is *0x8*, the next nibble has the size of the string and the next 7 bytes has the value saved in ASCII hexadecimal representation. Each character is store using 1-byte where the last character must be the end string character or *0x00*. Smaller strings are padded using the end character *0x00*. On the other hand, if the field needs to save a string with more than 6 characters a pointer is required. This pointer has an integer value that represent the ASL position file in the heap from the same entry or the heap from another entry. It points to a specific structure where it has a 4-byte field with the size of the string and then the required string plus the *0x00* character at the end.

Hence, the second part contains the pointers to the next entry, whereas the last field from the end part contains the pointer to the previous entry. Additionally, the second part contains the static fields and the size of the record without the first part. The third part contains at least 4 fields that if the string store is smaller than 7 characters it is stored in the field or if it is bigger it is stored in the first part of the entry. A schematic diagram is represented in figure 6.2 to be easier to understand how ASL works. A real hexadecimal representation of a ASL files is shown in figure 6.3 where a header and the first entry is explained using hexadecimal values:

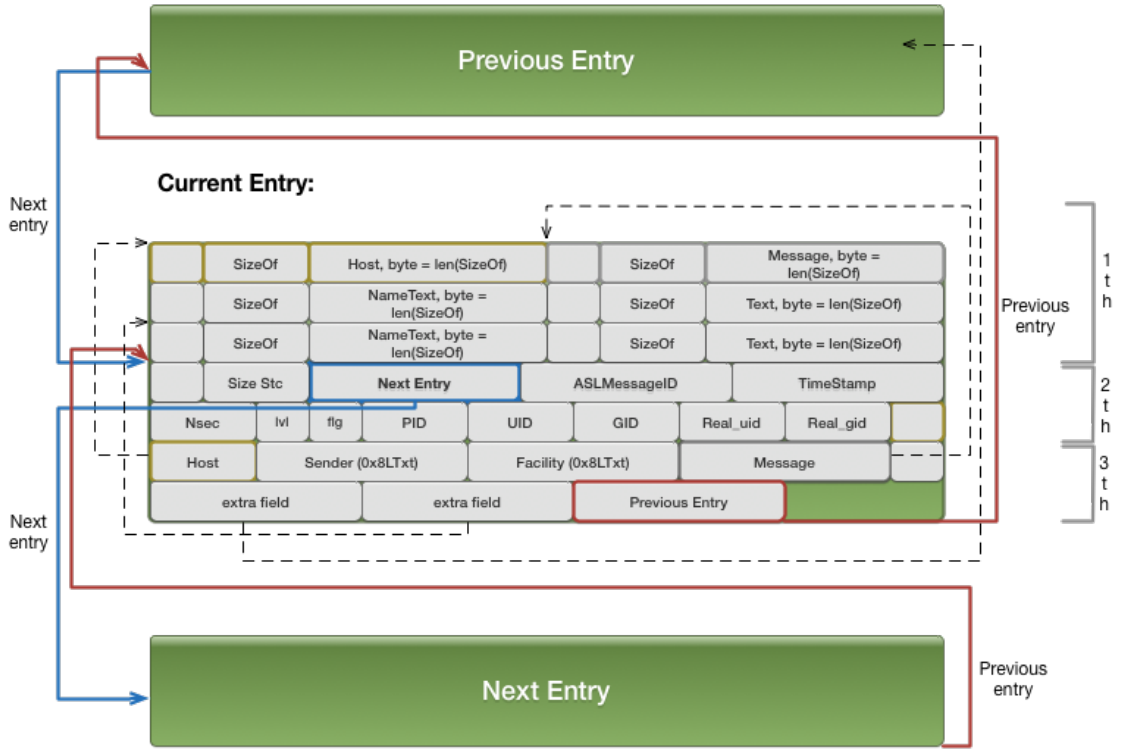

Figure 6.2: ASL Entry structure in deep.

The default ASL configuration indicates than the ASL binary format is used to store all the events coming from the process (PID = 0), in other words the kernel, and all the logs from Notice level (5) to Emergency level (0). This information is stored in the

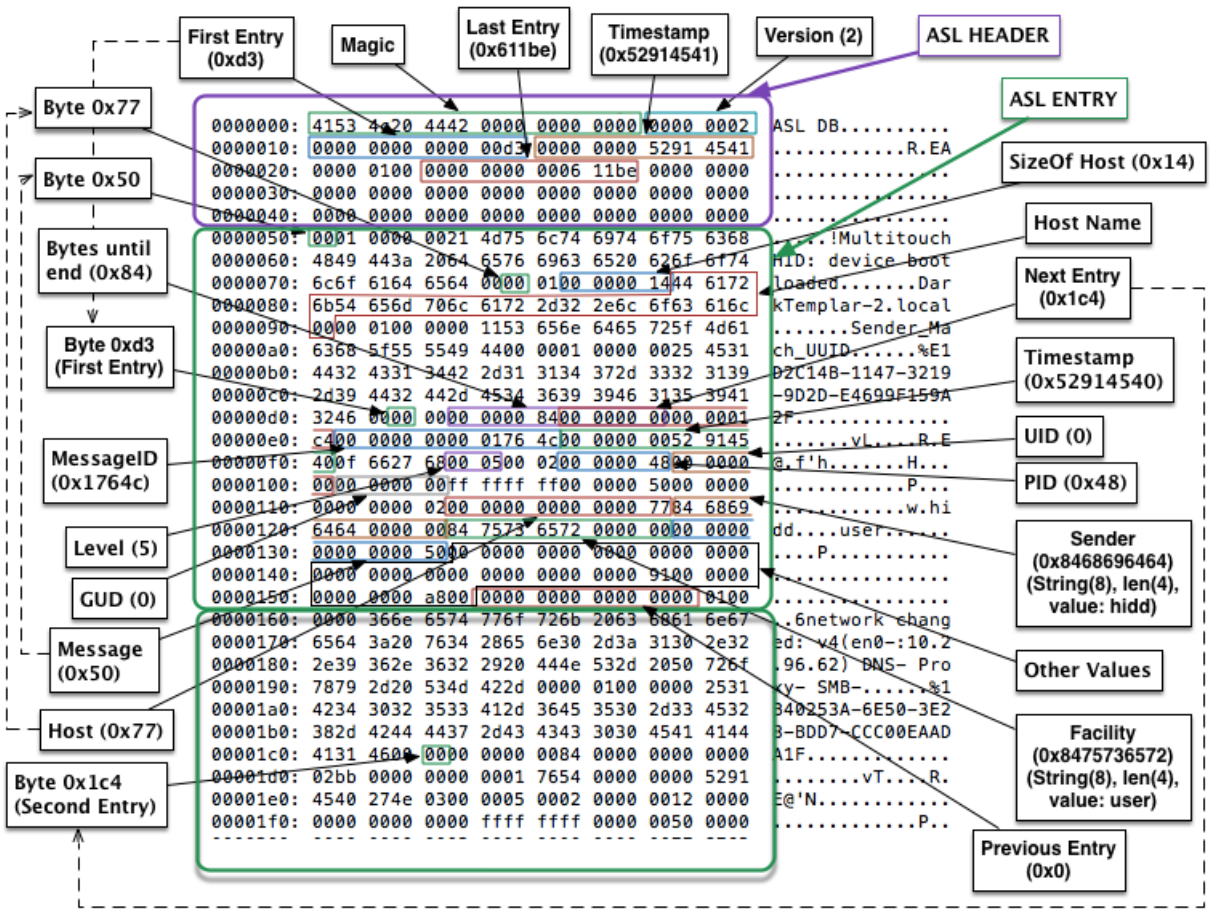

Figure 6.3: ASL Hexadecimal file example.

directory */private/var/log/asl*. These files contains useful information such as UTMP entries, external mounted devices such as USB or firewire devices, authentication user, application error or execution and so on.These files represents, after the file system evidence, the most important source of information from a forensics perspective.

Additionally, the binary ASL format is used to store other information in different directories. As an example, the diagnostic message is stored in the directory */private/-var/log/DiagnosticMessages* with the file format *YYYY.MM.DD.asl* and configured in */etc/asl.conf* or the power management that usually exists if the computer is a laptop where it stores all the information related with the battery of the computer in the directory */private/var/log/powermanagement* with the same file format than before and configured by */etc/asl/com.apple.iokit.power*.

Finally, and due to the fact that the document describes how ASL works, forensics should take into consideration the possibility that an attacker can be allowed to delete entries from these files. However, ASL binary format requires an extra work to do this task than the common binary files that store structures without relation between them. The reason is that not only is necessary to delete the information, also some modifications must be done over the header value and all the pointers from all the entries that stay below the entry that has been delete. Another approach instead of delete the information and change all the required fields, modify only the pointer from the previous entry and the next entry jumping the entry that wants to be hide. This is the common approach used by some malware to hide process modifying the process list, but instead of memory, it is done in the file.

A proof of concept has being implemented to be able to translate ASL binary files to plaintext without using Apple's tool where the source code can be obtained from Appendix A. Furthermore, it was implemented as a parser in Plaso and it is available since version 1.1 or Appendix B.

```
Path: "/private/var/log/asl/YYYY.MM.DD.[UID].[GID].asl".
Configure by: "/etc/asl.conf".
Format: ASL Binary file.
Example provided by the independent parsers documented in Appendix A:
        Record in: 0x212a
        * Next record in: 0x21d4
        * ASLMessageID: 160839
        * Timestamp: 2013-12-19 23:28:47 (1387495727)
        * Level: ERROR (3), PID: 48664
        * UID: 501, GID: 20, Read_UID: 501
        * Host: DarkTemplar-2.local
        * Sender: Google Chrome Helper
        * Facility: com.google.Chrome.helper
        * Message: CoreText CopyFontsForRequest received mig IPC error
            (FFFFFECC) from font server
        * CFLog Local Time: 2013-12-20 00:28:47.136
        * CFLog Thread: 507
        * Sender_Mach_UUID: 515D574D-2018-3D07-BF75-F125AAFE2591
```

6.1.2 Apple System Log plaintext format

Apple System Log can also store the data using plain text format called BSD format that it is the same format than the common Syslog such as Rsyslog or Syslogd, with the only difference that the message body can be stored using more than one line. For each registered event, ASL stores one entry. An entry is a plain text than can have one or more lines. The first part of the entry is the timestamp which is saved using the three first letters of the month, the day using the numeric value and the hour using the traditional *24 hour format: HH:MM:SS*. The second part starts with the hostname name that sends the information against the ASL follow by the sender, called also agent, which is stored using its name plus the process identification or PID between brackets, and finally, colon symbol to represents the end of the second part. Finally, the message itself that depending of the sender has different formats and sometimes required more than one to be stored:

Month	Day	HH:MM:SS	Host	Sender [PID]:	Message

Additionally, BSD plain text has a specific entry called repeated line indicating that the previous entry was repeated X times. In these repeated lines, we known when was the first time, when was the last time, how many times and the message information, but the timestamps between the first and the last entry are lost. This information is stored if the application uses the ASL binary format.

An example of a BSD plaintext entry stored using the ASL where it was an event on twenty-second of December at 00:02:46 sending by the Hostname *DarkTemplar-2.local*, the sender was *Chrome Helper* which its process ID was 65252 and the message was *CGSLookupServerRootPort: Failed to look up the port for "com.apple.windowserver.active" (1100)* is written bellow:

```
Dec 22 00:02:46 DarkTemplar-2.local Google Chrome Helper[65252]:
    CGSLookupServerRootPort: Failed to look up the port for
    "com.apple.windowserver.active" (1100)
```

One important drawback with this file format, and almost all the syslog than use plaintext format, is that they does not store the year when the event occurred, representing a challenge to create a valid time line. To obtain the year, the creation time or *crtime* of the HFS file attributes is obtained as a valid year for the first entry. Then, for each new entry the comparison between the month from the previous entry and the actual entry must be done. If the month of the actual entry happens before than the previous month it means that the log has moved to the next year, and then, the year must be incremented by one. As an example, if the previous entry was December (Dec) and the actual entry is January (Jan), first (Jan) is smaller than twelfth (Dec), then it means a new year. However, this approach is only valid if the timestamp difference between two consecutive entries is smaller than one year and if the crtime was not modified.

The PCRE regular expression to parse the grammar is as follow:

```
Month_Name = r'(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+'
Day_Number = r'(\d{1,2})\s+'
Time = r'([0-9]{2}:[0-9]{2}:[0-9]{2})\s?'
Hostname = r'(\S+)\s+'
Agent = r'([^:]+):\s+'
Message = r'([^\n]+)'
```

The most important evidences in BSD plain text format are stored in the directory */private/var/log/* with the file extension *.log*. Some examples are *system.log*, *authd.log*, *appfirewall.log* and *install.log*. Other important BSD files from the security perspective are stored in the directory */private/var/log/module/com.apple.securityd/* with the format *security.log.YYYYMMDDTHHMMSSZ*. The most relevant files have being documented below:

Authentication Log File: contains the authentication events coming from the sender *com.apple.authd* permitting to trace the user action that require authentication rights.

```
Path: "/private/var/log/authd.log"
Configure by: "/etc/asl/com.apple.authd"
Format: BSD plaintext.
Example: "Dec 22 15:18:53 DarkTemplar-2.local com.apple.authd[33]:
    Succeeded authorizing right 'system.disk.unlock' by client
    '/System/Library/CoreServices/CSUserAgent' [69667] for authorization
    created by '/System/Library/CoreServices/CSUserAgent' [69667] (3,0)".
```

System Log File: has all the events from the kernel and level less or equal to *notice*, represents the reduce version of */private/log/asl/** limited by a specific disk space.

```
Path: "/private/var/log/system.log"
Configure by: "/etc/asl.conf"
Format: BSD plaintext.
Example: "Dec 22 17:27:10 DarkTemplar-2.local WindowServer[114]: Display
    0x4273c00 released by conn 0x149ef".
```

Install Log File: contains all the events coming from the install facility, usually is used by the apple updates.

```
Path = "/private/var/log/install.log"
Configure by: "com.apple.install"
Format: BSD plaintext.
Example: "Dec 22 04:30:28 DarkTemplar-2.local softwareupdated (200)[297]:
    SoftwareUpdate: finished install of 031-1672".
```

Firewall Log File: contains all the events coming from the facility *com.apple.alf.logging* that correspond to the application firewall, level 7 firewall, included in Mac OS X.

```
Path = "/private/var/log/appfirewall.log"
Configure by:
Format: specific plaintext format.
Example: "Nov 2 17:05:39 DarkTemplar-2.local socketfilterfw[112] <Info>:
    Skype: Allow TCP LISTEN (in:0 out:1)".
```

Securityd: these group of logs have a specific plaintext format where it contains all the exceptions raise by the *Apple Security Module* [18] where it usually indicates the class and the method that generates the exception such as SSL and keychains issues.

```
Path = "/var/log/module/com.apple.securityd/security.log.YYYYMMDDTHHMMSSZ"
Configure by: "/etc/asl/com.apple.securityd"
Format: specific plaintext format defined in the configuration file.
"Time Sender[PID] <level> [Facility] SecAPITrace Caller: Message".
Example: "Dec 22 04:29:42 secd[195] <Error> [user{} ]:
    SecErrorGetOSStatus unknown error domain:
    com.apple.security.sos.error for error: The operation couldn't be
    completed. (com.apple.security.sos.error error 2 - Public Key not
    available - failed to register before call)".
```

Regarding to the implementation process of extraction these evidences, Plaso already had a Syslog parser than does not support the multiline body message and also without the repeat line translation. Hence, the first decision was modify the parser to be able to parse the multiline message body, and also, being able to translate the repeated message lines to the real message, where the previous message body must be obtained instead of the body of the repeated line entry. However, and after analyzed the results of the parser with the Plaso team, they noticed than the parser has a high performance impact being by far the slowest parser. For this reason, instead of modify the old parser, a new syslog parser has being implemented using the regular expression Pyparsing library to extract this data instead of the default regular expression in Python obtaining a speed improvement.

Furthermore, and also due to performance reason, two different Pyparsing parsers has been implemented. The first one called *syslog* is able to parse the plain text syslog, but only the first line of the message body is extracted omitting the other lines of the message body. The second syslog parser is called *syslog_mac* that it is able to parser the whole message body despite the message body has more than one line. However, this second parser is slower than the *syslog parser*. During the first phase, it is recommended to use the *syslog* parse and work with the results of this parser meanwhile the *syslog_mac* is working having a fast super timeline output to work meanwhile the full super timeline is created.

Additionally, other specific parsers have been implemented. As an example, the System Log File has a specific text format due to it has a message format between the PID and the body message. For this reason it was implemented as an independent Plaso parser called *mac_appfirewall*. Also, Securityd logs have their own format where the hostname is not required and in the middle of the log a structure *[Facility] SecAPITrace Caller: Message* is created. The *mac_securityd* Plaso parser has being implemented for *securityd* module evidences.

6.2 Basic Security Module (BSM)

Basic Security Module (BSM) was created initially for Solaris and SUN OS by SUN Microsystems to accomplish the C2 rating in the Trusted Computer System Evaluation Criteria (TCSEC) published by the US-NSCC, usually known as Orange Book. In contrast to ASL, which works in application layer, BSM is a kernel based auditing mechanism [77, 72].

Apple delegated the BSM implementation to McAfee Research. The project was released by Apple under the BSD license allowing to other operation system projects to add this BSM version to their environments such as FreeBSD since version 6.2 [96]. The recent versions of this implementation is maintained by the Trusted BSD Project, known as OpenBSM, where the last version is 1.1, also called 11 or *0xb*, the same version that uses the latest versions of Mac OS X.

BSM saves the information using different files where for each file it can have one or more than one entry. An entry is a binary structure that stores a kernel event. In other documentations this entry is also called trailer or track [10]. This entry is composed by a group of tokens where a token is a binary structure where depending of the type of token, the length of the structure can be static or dynamic (the length of the token is provided for one field in the token itself). Each token has a specific purpose representing a specific data: arguments of the program, return value, text data, socket, execution, action in a file, etc [69]. The tokens are stored using a big endian format. For each token, it has 8-bit unsigned big endian integer field that identify the type of the token follow by the structure of the token. Depending of the ID, the token structure is different.

Solaris and OpenBSD implementation indicates that only one token is mandatory for each entry: the header token [72] [93]. The token header contains the most important information such as the timestamp of the event, BSM version, the number of bytes of the entry (length) and identification BSM event, stored as an integer, that represent the type of the event itself. The auditors need to know the corresponding identification BSM number to its text explanation to understand the event. This mapping is done in the file *audit_event* in the directory */etc/security*. As an example, the BSM ID 45015 is translated as a *creation of a new group*.

Additionally, it is common to considered than the token subject (the information about the user who generates the entry) must be in each entry; however this is not mandatory. Also, the token should finish with two tokens: return token and trailer token. The first one indicates the return value provided by the process and the last

indicates the end of the entry as an extra control to separate one entry from another. Apple BSD follows this structure, but without being mandatory.

Hence, each BSM file can have zero or more events. Each event is a group of tokens where this event o entry starts with a token Header represented by ID 20, 21 and 116. Follow by zero or more tokens that provided an extra information, and finally, but not mandatory, the token Return represented by the ID 39, 82 and 114 and the token Trailer identified by ID 19 that represent the end of the entry as it is explained in figure 6.4.

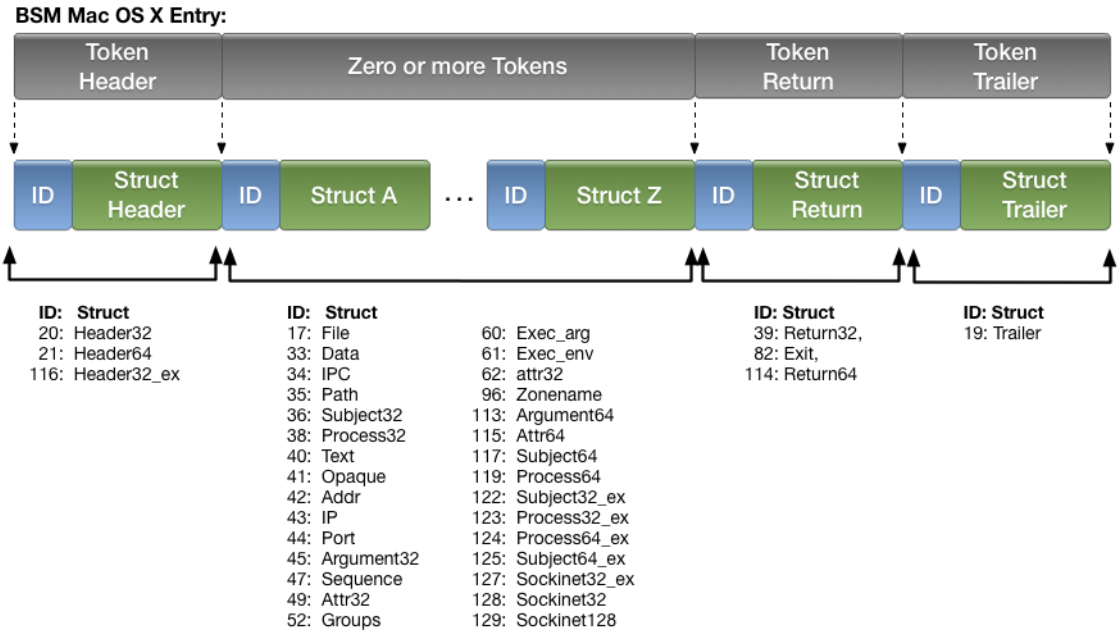

Figure 6.4: BSM Entry structure.

As an example the figure 6.5 is a Mac OS X BSM file that contains different entries. In the figure, the second entry is explained step by step. The entry has five tokens. The first token is the token header represented by ID 0x14 (32) that indicate a header type *Header32* that provided the timestamp, the length of the event (0x58, 88 bytes), the event type 45025 (Security Service AuthEngine) and identifying the BSM version as a 0xb (11 or 1.1). Following the header, the entry provides two extra tokens: Subject32 that indicate the user that did the action showing UID 0 and GID 0, then the root user, and the token Text that provides some text information: *begin evaluation*. Finally it provides the token Return32 that indicates than the event finish with status code 0 or correctly finished and the token Trailer that represents the end of the file.

The BSM files are saved in a group of files identify by *initial_timestamp.last_timestamp* where the timestamp are saved using an integer Epoch time. If the file does not finish correctly instead of using "last_timestamp" it uses the string "crash_recovery". Also the current BSM timestamp change the "last_timestamp" for the string "not_terminated". These files are saved in the directory */private/var/audit/starttime.endtime* where only the root user can have access to these files.

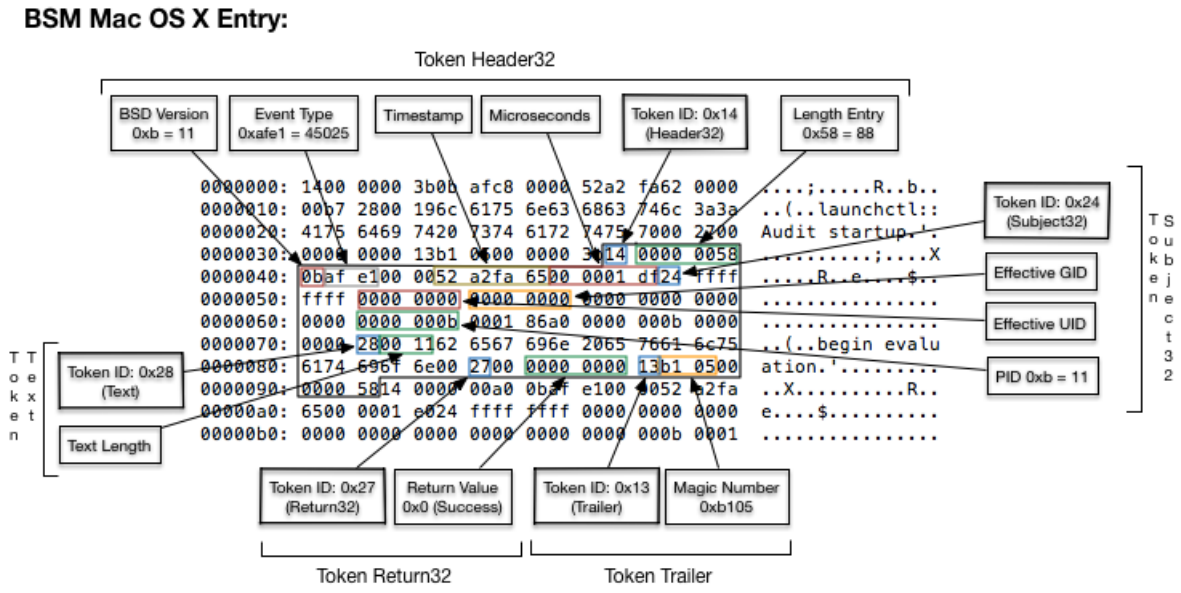

Figure 6.5: BSM hexadecimal raw entry.

Mac OS X provides the tool *Praudit* to read these entries permitting that the output can exported as a plain text or XML format to be analysed by the auditors [32]:

```
XML format:
$ praudit -e bsm_file

PlainText:
$ praudit bsm_file
```

Regarding to the extraction process of these evidences for forensics purposes, the auditors need to know all the token IDs and their structure associate with the token ID. However, BSM has different version due to the operation system evolution, 32 and 64 bits architectures, extended fields with different options between structures and network evolution: IPv4 to IPv6. As an example, the documentation provided by Oracle [69] cannot be used to understand the Mac OS X version because it works for the initial Solaris BSM implementation; nevertheless the Oracle documentation provides useful information for the majority of the tokens.

To analyse the BSM structure, the official Apple documentation [14] with the Oracle documentation have combined. Taking into consideration that the information provides by Apple is not enough clear, has some blanks issues or it is out of date because it is focus on the understanding how BSM works and how to use the *Praudit* tool instead of how the information is stored. Finally, having the assurance that the tokens will be parsed correctly, the Apple XNU source code is used to verify the correct implementation, especially the Audit source code [9] and the BSM source code [10] .

Furthermore, the source code of OpenBSM [93] is used to identifying all the possible implemented structures. The source code package from OpenBSM provided a directory test with a group of fifty records, where only eighteen are different tokens, that has been used to validate the BSM parser comparing the output provided by the parsers with the results provided by *Praudit*. After the OpenBSM, Apple BSM and Oracle BSM research around forty different tokens are identified. However, in the OpenBSM header source code, it has defined close to hundred different structures, despite its implementation only support forty different structures. Moreover, only around twenty different token structures can be checked against a real Apple BSM or OpenBSM test entries.

For this reason, Plaso parser works in three different status. The **trusted status** refereed to the ones that have been checked against real Mac OS X BSM or OpenBSM files. The second status is called **not checked or untested** token structures where the implemented structures are obtained from the OpenBSM source code [93] minus the structures that already have being implemented as a trusted status. The last status is called **unknown** structures that represents structures that are not implemented in OpenBSM project.

For each entry or event in the file, the parser starts in the trusted status reading the header token obtaining the length of the entry. Knowing the length of the entry, it parses the rest of the entry using only the well known structures or trusted status. When it finds a no well know structure, it tries to use the combination between the well know structures and untested structures. If after parsing the entry, it arrives to the end of the entry provides by the header token, it assumes that the parse was correct, and then, the untested structures is well implemented. However, if the Token ID is unknown or the parser finishes before or after the length provided by the header, it assumes that the parsing was incorrect, getting all the information of the entry before this unknown or untested token ID, notifying that the entry which token ID in the byte file position is unknown and jumping to the next entry.

This implementation provided defenses against unknown structures, new implementation structures, some anti forensics modifications and file errors. However, the parser avoids these errors only if the altered or unknown tokens are not the header because the header provided the basic information to be able to parser the entry (length of the record). Additionally, the untested structures implementation was tested against an expected input using a python script that check if the implementation is correct, despite it was not able to test it against real evidences. The table 6.2 shows all the implemented structures, its Token ID and if it was tested against a real BSM file.

The Plaso implementation with all the BSM structures can be found in *Appendix B* or in the PLASO project since 1.1, or later. Furthermore, a proof of concept script has been provided in the Appendix A. Both parsers were able to parse all the OpenBSM evidences and BSM evidences coming from seven different Mac OS X 10.8 and 10.9 environments without found any error or any unknown structure. The list of implemented tokens with their structures is explained in Appendix C.

Token ID	**Token structure**	Tested
17	BSM_TOKEN_FILE	Yes
19	BSM_TOKEN_TRAILER	Yes
20	BSM_HEADER32	Yes
21	BSM_HEADER64	Yes
33	BSM_TOKEN_DATA	Yes
34	BSM_TOKEN_IPC	Yes
35	BSM_TOKEN_PATH	Yes
36	BSM_TOKEN_SUBJECT32	Yes
38	BSM_TOKEN_PROCESS32	Yes
39	BSM_TOKEN_RETURN32	Yes
40	BSM_TOKEN_TEXT	Yes
41	BSM_TOKEN_OPAQUE	Yes
42	BSM_TOKEN_ADDR	Yes
43	BSM_TOKEN_IP	Yes
44	BSM_TOKEN_PORT	Yes
45	BSM_TOKEN_ARGUMENT_32	Yes
47	BSM_TOKEN_SEQUENCE	Yes
49	BSM_TOKEN_ATTR32	No
50	BSM_TOKEN_IPC_PERM	No
52/59	BSM_TOKEN_GROUPS	No
60	BSM_TOKEN_EXEC_ARGUMENTS	No
61	BSM_TOKEN_EXEC_ENV	No
62	BSM_TOKEN_ATTR32	No
82	BSM_TOKEN_EXIT	No
96	BSM_TOKEN_ZONENAME	Yes
113	BSM_TOKEN_ARGUMENT64	Yes
114	BSM_TOKEN_RETURN64	Yes
115	BSM_TOKEN_ATTR64'	No
116	BSM_HEADER32_EX	Yes
117	BSM_TOKEN_SUBJECT64	No
119	BSM_TOKEN_PROCESS64	Yes
122	BSM_TOKEN_SUBJECT32_EX	Yes
123	BSM_TOKEN_PROCESS32_EX	No
124	BSM_TOKEN_PROCESS64_EX	No
125	BSM_TOKEN_SUBJECT64_EX	No
126	BSM_TOKEN_ADDR_EXT	No
127	BSM_TOKEN_SOCKINET32_EX	Yes
128	BSM_TOKEN_SOCKINET32	Yes
129	BSM_TOKEN_AUT_SOCKINET128	No
130	BSM_TOKEN_SOCKET_UNIX	Yes

Table 6.2: BSM implemented token structures.

6.3 UTMPX File

Traditionally, the Unix systems have stored the register of the user logins, active user and last logins using the binary files UTMP, WTMP and Lastlogin. However, since Mac OS X 10.5 [19] these logs have been deleted storing this information using the Apple System Log in binary format. Nevertheless, and for backwards compatibilities, the UTMP still exists, but it has been replace by new version called UTMPX. The file is stored in */private/var/run/utmpx*.

The UTMPX file is a binary file that contains a group of entries. Each entry represent active and inactive sessions since the system was booted providing when these accounts were logged in the system, the username, the type of terminal was used, from witch hostname, the identification process, etc. The entry structure is provided by the Apple documentation [3] where the UTMPX Mac Os X structure has some difference comparing with the standard BSD structure. Despite the previous Apple structures such as ASL or BSM, UTMPX stores the information in Little Endian instead of Big Endian. The Apple documentation uses *UTX* constants and specific types that they can be obtained from the Apple source code [8]. The structure researched in Mac OS X environments uses 2 bytes more than it is documented due to the *short ut_type issue* (explained below). Hence, the real size of each entry is 628 bytes. Where the most important fields in this structure are below:

Order	Name	Length
1th	User name	256 bytes
2th	ID size	4 bytes
3th	Terminal name	32 bytes
4th	Process ID	4 bytes
5th	Session status	*4 bytes
6th	Epoch timestamp	4 bytes
7th	Microsecond	4 bytes
8th	Host name	256 bytes
9th	Padding for future fields	64 bytes

Table 6.3: UTMPX structure.

- **ut_user** 256-byte field that save the name of the user as a hexadecimal representation of the ASCII character. If the name of the user is less than 256 bytes, the right part of the field is padded by '0x00'.
- **ut_line**: 32-byte field that save the terminal name as an hexadecimal representation of the ASCII characters. If the name is less than 256 bytes, the right part of the field is padded by '0x00'.
- **ut_type**: this field stores the numerical representation of the terminal status. Following the Apple documentation the field is a short type variable. Hence, it is suppose to use only 2 bytes [3]; however, and after analyze different UTMPX files,

the behaviour is different (Figure 6.6). Between the *pid_t ut_pid* that it is a 4-byte field and the *struct timbal ut_tv* that it is a 8-byte field, instead of being a 2-byte field that represents the terminal type, a 4-byte field is observed. For this reason, and despite the Apple source code, two explanation can be assumed: *ut_type* is a 4-byte field or it is a 2-byte field plus 2-byte of padding:

```
Original Apple source code:
struct utmpx {
        char ut_user[_UTX_USERSIZE];    /* User name, 256 bytes     */
        char ut_id[_UTX_IDSIZE];        /* 4 bytes                  */
        char ut_line[_UTX_LINESIZE];    /* Terminal name, 32 bytes  */
        pid_t ut_pid;                   /* 4 bytes                  */
        short ut_type;                  /* Session status, 2 bytes  */
        struct timeval ut_tv;           /* Timestamp, 8 bytes       */
        char ut_host[_UTX_HOSTSIZE];    /* 256 bytes                */
        __uint32_t ut_pad[16];          /* 64 bytes, 16 * 4 bytes   */
    };
```

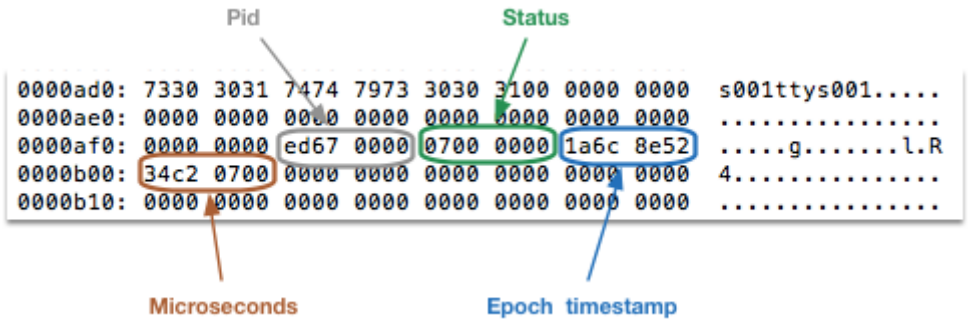

Figure 6.6: Real UTMPX Mac OS X structure

The *ut_type* field can get different values [34], where the most common are 0x07 (user process) that represent an active user session and 0x08 (dead process) that represents a closed user session.

- **ut_tv**: is a *timeval* structure (*timeval.h*) with two 4-byte integers that provides the Epoch timestamp format of the entry when the session was started. The first four bytes are in a little endian representation and it represents the Epoch time in seconds. The next four bytes are also in little endian and it provides the microsecond.
- **ut_host**: contains the host name or IP from the original server that the user comes from. If the user comes from the host itself (local user) this field is padded by 0x00.

When the system is booted the UTMPX file is created with two entries. The first

entry is an empty entry with only two fields filled out: the username and the type (*ut_type*. The username is *utmpx-1.0* and the type is a Darwin unique type called *Signature*. This first entry can be considerer as a header or magic value for UTMPX Mac OS X file. The second structure is created when the boot process started where the type is *boot time*.

Each time the system is started, the file UTMPX is cleaned and a new file is created. Each time the computer is halted the type session is modified. As an example, the user processes (0x07) are closed, and then, they are changed to dead processes (0x08). For this reason, if during the evidence acquisition the computer is shutdowned instead of being halted, the UTMPX should be copied before.

An UTMPX parser is provided as a proof of concept in the appendix A. Additionally, a Plaso parser is implemented since 1.1 documented in Appendix B. Furthermore, in the same Plaso version, an UTMP Linux parser file was implemented where the difference between the Linux structure and Mac OS X structure can be analysed.

6.4 Configuration Files: Property List

Mac OS X uses the Property List (Plist) files to store values usually related with configurations. It can be showed as the registry in Windows environments. As it was indicated in previous chapters, these files can be stored using two formats: XML or Binary Property List format. Each evidence contains a group of attributes where only a few of them are useful in the forensics context. Also, some of these attributes are another XML or Binary Plist structure or an unknown binary data structure that it must be parsed.

Related to parsing process, the binary plist python library called Binplist [70] was used. This library was implemented in middle of 2013 providing a full support to Binary Plist except the version 1.6; Nevertheless this version is not used in our Mac OS X evidences. The library has some issues to parse Binary Plist when the file contains attributes that contains itself another Plist structure (XML or binary) where it is not able to parse the structure properly, returning a binary representation of the plist structure. In case of XML Plist files, the official Python library *plistlib* [35] was used.

Neither libraries are able to parse the specific binary attributes plist structures such as *bookmark* or *BackupAlias*, where a research, reversing engineering and Construct structure need to be implemented to extract this information. The tools called *mac_plist.py, alias.py and recent.py*, documented in Appendix A, are an independent parsers that can extract all the documented evidences explained in this section. In Plaso, there were implemented as a plugins inside the *plist_plugin engine*, documented in appendix B.

6.4.1 Time Machine

The first evidence is the file **com.apple.TimeMachine.plist** that contains all the basic information related to the backup Apple tool called Time Machine. This file contains all the hard disk used with this tool: UUID, alias and all the times when the

back up was done. Furthermore the attribute *BackupAlias* contains a binary data that store information about the hard disk used to store the backups. The name assigned by the user to this hard disk is stored in the eleventh byte of this binary structure. At eleventh byte it has the length of the text followed by the text name that store the alias name:

```
Directory: /Library/Preferences
Plist attributes:
Destinations -> For each item
    DestinationUUIDs = Identification device where the backup is done.
    SnapshotDates = list of items with the all the backup timestamp.
    BackupAlias = hard disk alias name (binary structure).
```

In case that the auditors have also access to the Time Machine device, the tool *time dog* [86] can be used to check the differences between Time Machine backups.

6.4.2 Bluetooth

The next evidence is **com.apple.Bluetooth.plist** that contains all the bluetooth devices that the computer has detected and connected. It provides the MAC of the device, some times the name of the bluetooth device and the timestamp from the last time that this device was detected.

```
Directory: /Library/Preferences
Plist attributes:
DeviceCache -> For each deviceCache
    "MAC Device"
      Name = Bluetooth name (optional field).
      LastInquiryUpdate = timestamp.
```

6.4.3 Mac OS X Updates

Another evidence is **com.apple.SoftwareUpdate.plist** that contains the last timestamp when the Mac OS X was partial and fully update. Also, it can provide the version of the Mac OS X, the number of pending updates, the name of these updates and the version.

```
Directory: /Library/Preferences
Plist attributes:
    LastSuccessfulDate = timestamp from the last partial update.
    LastFullSuccessfulDate = timestamp from the last full update.
    LastAttemptSystemVersion = last Mac OS X version.
    LastRecommendedUpdatesAvailable = number of pending updates.
```

```
    RecommendedUpdates = array with the pending updates.
```

6.4.4 Airport Stored Wifi

The next evidence is the file **com.apple.airport.preferences.plist** that contains all the stored wireless, providing useful information such us the SSID and the security protocol. Also, the file stores the last time that the wireless was used.

```
Directory: /Library/Preferences/SystemConfiguration/
Plist attributes:
RemeberedNetworks -> For each item.
    LastConnected = timestamp.
    SSIDString = string representation of the wireless name.
    SecurityType = String with the security value.
```

6.4.5 Spotlight Searched Terms and Volume

Spotligth is a tool, known as magnificent toolbar that permits search applications or file only writing a short description about what we are looking for. As an example, if we write the word "pdf" it shows the last open PDF files in the computer. The Plist **com.apple.spotlight.plist** stores a history usage from this tool providing all the searched terms, the last time when this term was search, which was the selected value by the user and the real path to this value.

```
Directory: /Users/user/Library/Preferences/
Plist attributes:
UserShortcuts -> For each item
    "StringSearch"
        PATH = Path of the file or application.
        LAST_USED = Timestamp from the last time it was used.
        DISPLAY_NAME = String shows by Spotlight.
```

Additionally, the plist file **VolumeConfiguration.plist** contains the volumes that are activated to be used with Spotlight. It provides the information identification of this volume (UUID), where it was mounted and the timestamp when it was activated:

```
Directory: /.Spotlight-V100/
Plist attributes:
Stores -> For each UUID Volume
    CreationDate = Timestamp when the volume was activated.
    PartialPath = Path where the volume is mounted.
```

6.4.6 Apple User Account

Another file is **com.apple.coreservices.appleidauthenticationinfo.UUID.plist** that provided information related with the Apple account such as the AppleID, the name and the family name associated to this account, the first time when the account was configured, the last time when the account was authenticated, etc.

```
Directory: /Users/user/Library/Preferences/ByHost/
Plist attributes:
Accounts -> For each account
    "AppleIdAccount"
        AppleID = identification user from Apple account.
        FirstName = First name from the account owner.
        LastName = Family name from the account owner.
        AccountUUID = Identification from the Apple account.
        CreationDate = when the account was configured the first time.
        LastSuccessfulConnect = tlast time that it was connected.
        ValidationDate = last time it was validated.
```

6.4.7 System User Account

The plist files **user_name.plist** was explained in previous chapters. The file contains the system user information where the *passwordpolicyoptions* attribute that contains a second XML plist structure embedded inside the plist file. This structure contains three timestamp attributes and one information attribute.

```
Directory: /private/var/db/dslocal/nodes/Default/users
Plist attributes:
    passwordLastSetTime: last time the password was changed.
    lastLoginTimestamp: last time the user was logged into the system.
    failedLoginTimestamp: last time the user introduced an incorrect
        password.
    failedLoginCount: number of consecutively times that the user
        introduced an incorrect password.
```

The three timestamp attributes from *passwordpolicyoptions* stores their times using a COCOA timestamp base. Some of these values are reset to COCOA timestamp 0 (2001-01-01 00:00:00) if the status that they store has changed without maintain a log. As an example, if the user attempts to login into the system with an incorrect password the timestamp from *failedLoginTimestamp* is set and the failedLoginCount is incremented in one. When the user introduces the valid password, the timestamp and the counter are reset to 0.

6.4.8 Installation History

The file contains all the previous installed program in the system, the name of the program, the version, how it was installed and when it was installed. This information is only stored when the program was installed using a Mac OS X installer, DMG files, PKG files, etc. Then, if a malicious user or malware copy or download a program and stored it in a specific directory, this installation is not revealed in the InstallHistory file.

```
Directory: /Library/Receipts/InstallHistory.plist
```

6.4.9 Finder Sidebar

The Mac OS X GUI file system browser called Finder uses a plist file to store the history of mounted devices. The plist contains the configuration and history of the Finder sidebar. Regarding with forensics purposes, the *systemitems* element, and inside this, *the VolumesList* element contains the list of mounted devices. Each element of the list contains two important attributes. The first attribute called *Name* and it is the name showed by Finder whereas, the second element is a binary structure called *Alias*. This second attribute is a binary structure that provided useful information such as the volume name, the mount point. If the mounted device was a DMG or HFS file system it contains a integer HFS timestamp with the date when the partition was created. However, since Mac OS X 10.9 the DMG files do not stored the timestamp where only the mounted external devices formatted as HFS are stored in the sidebar property list.

```
Directory: /Users/usr/Library/Preferences/com.apple.sidebarlists.plist
```

The Alias attribute is not documented by Apple. It is a big endian binary structure that has been reversed partially. The header contains a 2-byte field with the length of the Alias field follow by two fields of 4-byte that contains an integer timestamp in HFS time. If the file is not a DMG or HFS device the value is 0. Both timestamp have always the same value and the reason to have two different timestamp is unknown. Follow the Alias header, the field has different attributes where the first 2-byte represents the type of the attribute. If the type value is 0x0000000e (14) it is the VolumeID with the name of the volume whereas if the value is 0x00000013 (19) it contains the mount point of the device. The VolumeID contains two identical structures where the first structure is the name showed by the Finder and the second attribute is the name of the Volume. The attribute is 2-byte field with the volume length follow by 2-byte field that represents the number of characters of the string follow by an UTF-16 (2-byte per character) string that contains the name. Regarding to the mount point, it is a structure that contains the path where the device was mounted. It is a 2-byte structure with the number of characters of the string follow by an UTF-8 string that contains the mount point path. An hexadecimal explanation is explained in fire 6.7.

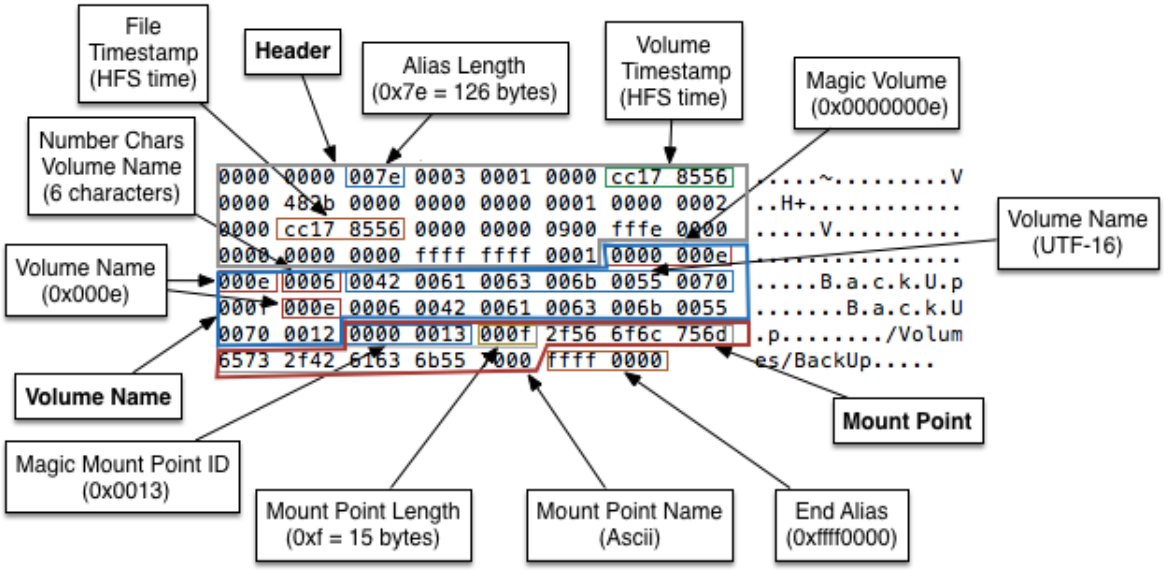

Figure 6.7: Hexadecimal explanation of alias attribute.

Finally, this information can be only used to have a list of mounted devices because the timestamps is not when the device was mounted, instead it is when the device was created or formatted by the creator. In case of work with Mac OS X 10.8 or lower, the timestamp of the DMG files can provided the version of the program if the timestamp is compared over the different DMG version of the program.

A proof of concept parser called *alias.py* has being implemented and the output over a proper list from a Mac OS X 10.8 can be showed in figure 6.8. As we can see in the picture, the DMG file *wireshark* contains the HFS timestamp because it comes from a Mac OS X 10.8, however the value will be 0 if it comes from a Mac OS X 10.9. Furthermore, some devices such as *Zeratul* or *Peliculas* have a 1904-01-01 timestamp (0 in HFS) because they are Fat32 and NTFS devices that do not store the timestamp.

6.5 Newsyslog plaintext logs

As it was described before, Mac OS X uses the Apple System Log to store the applications logs. However, some specific applications have their own file logs instead of use an external service to store their logs. This decision creates system space issues and performance impact due to big file logs than are increased constantly. To avoid these problems, Mac OS X uses an external tool called Newsyslog from FreeBSD to rotate these files.

Reading the configuration file from Newsyslog several numbers of these files can be identified. The default configuration file is */etc/newsyslog.conf* and other extra configurations are in the directory */etc/newsyslog.d/*. The logs that Newsyslog rotates are stored in the directory */Library/Logs/* with the extension dot log (*.log*). The only excep-

```
Name: BackUp2
Time: 2013-09-14 21:40:56
Volume name: BackUp2
Volume time: 2013-09-14 21:40:56
Mount point: /Volumes/BackUp2

Name: Peliculas
Time: 1904-01-01 00:00:00
Volume name: Peliculas
Volume time: 1904-01-01 00:00:00
Mount point: /Volumes/Peliculas

Name: ZERATUL
Time: 1904-01-01 00:00:00
Volume name: ZERATUL
Volume time: 1904-01-01 00:00:00
Mount point: /Volumes/ZERATUL

Name: Wireshark
Time: 2013-09-10 18:01:29
Volume name: Wireshark
Volume time: 2013-09-10 18:01:29
Mount point: /Volumes/Wireshark
```

Figure 6.8: Extracting mounted devices extracted using *alias.py* script.

tion is the log coming from the Wi-Fi device which is stored in */private/var/log/wifi.log*. Indeed, and after research these evidences, the **Wifi.log** is the only log from the rotate logs by Newsyslog that provided useful information.

Wifi.log stores all the Wi-Fi activity. Newsyslog limits the file space to 2.5 megabytes, where for an average computer uses it permits to store the last two months of Wi-Fi activity. This file contains useful information such as when the wireless was turn up, when the wireless was connected and providing the name of the wireless, the properties of the Wi-Fi when it was configured for the first time: SSID, Security, BSSID, channel, etc. The wifi.log stores its information using a plaintext with some differences if we compare with the basic BSD plain text format: the first word in the log entry is the first three letters of the week day, it does not save the hostname and the sender is written between the less and greater symbols.

Week Day	Month	Day	HH:MM:SS	Sender [PID] :	Message

The grammar to parse these files is explained below:

```
Day_Name = r'(Mon|Tue|Wed|Thu|Fri|Sat|Sun)\s?'
Moth_Name = r'(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+'
Day_Number = r'(\d{1,2})\s+'
Time = r'([0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{3})\s?'
Sender = r'\<([^\>]+)\>\s+'
Function = r'([^:]+):\s+'
Message = r'([^\n]+)'
```

A two entries from the wifi.log example is provided below:

```
Thu Nov 14 21:52:09.883 <airportd[88]> _processSystemPSKAssoc: No
    password for network <CWNetwork: 0x7fdfe970b250> [ssid=AndroidAP,
    bssid=88:10:1a:7a:f1:58, security=WPA2 Personal, rssi=-21,
    channel=<CWChannel: 0x7fdfe9712870> [channelNumber=11(2GHz),
    channelWidth={20MHz}], ibss=0] in the system keychain
Sat Dec 21 06:16:56.168 <airportd[104]> _doAutoJoin: Already associated
    to ?DarkTemplar?. Bailing on auto-join.
```

The parser *mac_wifi* was developed in Plaso to parse the file identifying the most relevant events from the message such as new configured Wi-Fi (*processSystemPSKAssoc: plus in the system keychain*), wireless device turn on (*airportdProcessDLILEvent: InterfaceName attached (up)*) and Wi-Fi connections (*doAutoJoin: Already associated to SSID_name*). The source code is available since Plaso version 1.1. and documented in Appendix B. Also, an independent parsers can be found in the Appendix A, as an example of a parser using the regular expression described before.

6.6 CUPS IPP control files

Mac OS X uses CUPS as a printer daemon such as other Unix and Linux operation system. If the printer is not a local printer, CUPS uses the Internet Printing Protocol (IPP) to manage the remote printers. When this service is used, CUPS save in the directory */private/var/spool/cups/* one file for each job that it has printed ordering the jobs by integer value. The files have the syntax "cXXXXX" where this XXXXX represents the integer value of the printed job, then, the 00001 represents the first job, 00002 the second, etc. These files are called control files.

The control files can provided information about when the document was printed, which application was used to print it, the type of the document, how many copies, the name of the printer, the URI, the owner and user of the document, the host name, the job ID and the job name, etc. However, it does not provided the name, the path of the original file and the content of the original file. Nevertheless, if the document was printed using the PostScript driver, it creates a copy of the printed document in the same directory than the control file using the same numerical control number, but instead of starting with the letter C, it starts with the letter D [89].

The files are stored using a binary IPP structure [28] in Big Endian format. The first 8 bytes of the file is the header where the two first bytes represent the version: the first byte represent the major version and the second the minor version. The third and fourth are the IPP operation ID and the last 4 bytes are the IPP request ID. The latest versions of Mac OS X (10.8 and 10.9) implements IPP version 2.0. After the header, the file might have one group attributes or more and finally the file finish with the value "0x03" that represented the group ID END.

The Group attributes starts with one byte that represents the identification value of the group attributes and one or more attributes. The identification group attribute value

can be 0x00 to 0x07 except 0x03 that it is assigned to the end of the file. As an example 0x01 represents the operation attributes (charset, language, etc.) and the 0x02 the job attributes. Each group may have a list of zero or more attributes where an attribute is a pair of name - value. The name represents the text name of the attribute and the value is the value associate to this name. The first byte of this attribute is the identification attribute that indicates the type of value: integer, range of integer, resolution, text, date times or boolean [75]. In case of Mac OS X the timestamp values are stored using an integer value instead of date times value. The binary format is explained in the figure 6.9 where the raw representation is explained in the figure 6.10.

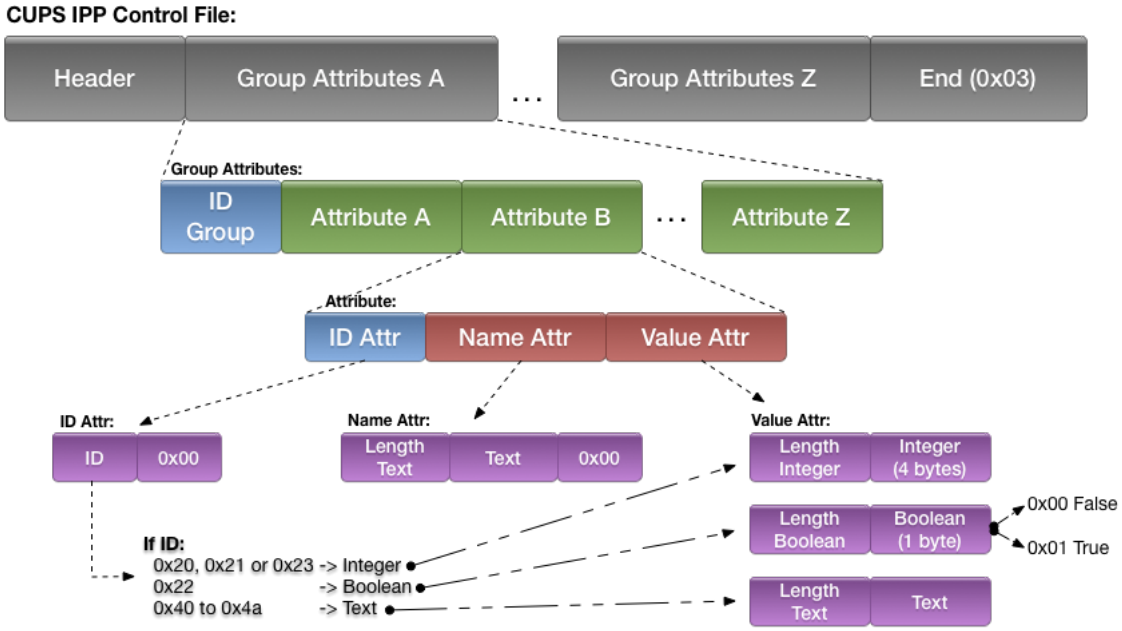

Figure 6.9: Diagram CUPS IPP control file.

Each file represents one job. Each job contains three important attributes that provided the timestamp values: when the job was created (time-at-creation), when the job was processed by the printer (time-at-processing) and when it was finished (time-at-completed). Hence, for each file that represented a unique job, the parser adds three different entries to the super timeline. Additionally, IPP stores so much information that it is important concerning to the printed job, but not for the forensics purposes. For this reason in the table 6.4 the most important attributes in Mac OS X IPP are indicated.

An example of the output from one control CUPS file can be read below using the independent CUPS IPP parser where its code is documented in Appendix A. Also, it was implemented in Plaso where its code is available since Plaso version 1.1 or late and documented in Appendix B. Finally, mention that exists a Python library called Pkipplib [1] that can parser these control files; however it was not updated since 2006 and the code is a little bit disorganized or mess, for this reason an own parser and Plaso plugin has written using the Python library Construct.

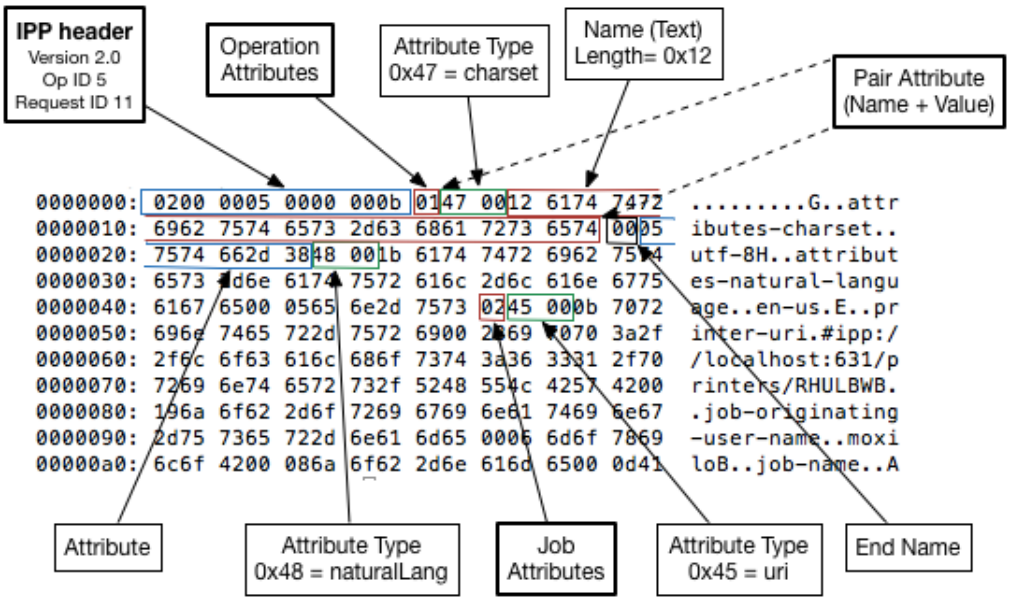

Figure 6.10: Hexadecimal CUPS IPP control file.

Name Attribute	**Value**
time-at-creation	When the job was created
time-at-processing	When the job starts to be processed
time-at-completed	When the job was finished
printer-uri	The printer URL
copies	The number of document copies
job-originating-user-name	System user name
job-name	Name of the job
document-format	Type of document
job-originating-host-name	Computer name
com.apple.print.JobInfo.PMApplicationName	Application used to print
com.apple.print.JobInfo.PMJobOwner	Completed name of the user

Table 6.4: Most valuable CUPS attributes.

```
# python cups.py c00001
Creation time: 2013-11-03 19:07:21 (1383502041).
Process time: 2013-11-03 19:07:21 (1383502041).
Completed time: 2013-11-03 19:07:32 (1383502052).
URI: ipp://localhost:631/printers/RHULBW
User: moxilo
Job name: Assignament 1
Application: LibreOffice
Owner: Joaquin Moreno Garijo
Copies: 1
Printer ID: RHULBW
Job ID: urn:uuid:d51116d9-143c-3863-62aa-6ef0202de49a
Computer name: localhost
Document format: application/pdf
```

6.7 Document Versions

In Mac OS X 10.7, Apple released a functionality that permits compare different versions over the same file [17]. As an example, if the user is writing a text document, every time this user stores the file, Mac creates a copy of this file permitting the user check the different versions of this file. This feature is called *Document Versions* or *Versions*.

Document versions is stored in the directory */.DocumentRevisions-V100*. The Document version directory contains another directories. The most important is the directory *PerUID* that contains one directory for each user that has used this functionality. The name of this directory is the Mac OS X user identification (User ID or UID). As an example, if the user with ID 501 has stored a text document, the copy file was stored inside the directory */.DocumentRevisions-V100/PerUID/501/*.

Each user directory ID contains a group of directories where the name is a two characters hexadecimal value "[0-9a-f][0-9a-f]". Inside each of these directories the information is classified by another directory where the name is the client that store the information (always com.apple.documentVersions). Finally, inside this client directory, a rename file with a UUID identification name plus the original file extension. As an example, if an user with ID 501 that has stored a PDF file, the Document versions will create a copy of this file called *CA088CEE-8828-43D2-AE67-3B873F695BCE.pdf* in the directory:

```
/.DocumentRevisions-V100/PerUID/501/1d/com.apple.documentVersions/
```

Document Version uses an SQLite file *db.sqlite* in the directory */.DocumentRevisions-V100/db-V1* to know the relationship between the original file and the stored versions of this file. Providing information such as the timestamp when the version copy was

done, the last time that this file was opened, the original file inode, etc. The SQLite query that it is required to extract this information from the *db.sqlite* file is documented below:

```
SELECT f.file_name AS original_name, f.file_path AS original_path,
f.file_last_seen AS last_time_checked, g.generation_path AS version_path,
g.generation_add_time AS version_time
FROM files f, generations g
WHERE f.file_storage_id = g.generation_storage_id;
```

The Document Version Plaso parser was coded with the name *mac_document_versions.py* using the SQLite engine, where the code is available since Plaso Version 1.1 documented in Appendix B.

6.8 Keychains

Apple provides a password mechanism protection to store the credentials [57] except the system user credentials than are stored using a property list explained in the previous chapter. Keychain stores the required credentials for the applications, network connections, email accounts, web site, public/private key, certificates, etc. Apple provided the Keychain API to store safety the password of the user avoiding that each application requires implementing their own solution. In Mac OS X, the operation system uses one keychain file to store safely the credentials required by the operation system itself, as an example the wireless password, and one additional keychain file for each user in the system [56].

```
System: /Library/Keychains/System.keychain
User: $home/Library/Keychains/login.keychain
```

Keychain uses a 3-DES block cipher algorithm for encrypt [13] the information; However, only the password and the secure notes are encrypted, all the other information is not. Then, a forensics analyser that get access to one of these files is able to obtain useful information despite the password to decrypt the information is unknown. As an example, the keychain file from an user can provide all the URLs where the user stores the password, when the password was stored for first time, the last time the password was changed, the username used in that webpage, the URL of the webpage, etc. Taking into account that not all the applications use the keychain to store the credentials. Then all this information can be obtained only by the applications use the keychain API to store safely the password. As an example, Google Chrome or Apple Safari uses keychain platform to store this information, whereas Mozilla Firefox uses its own database.

The keychain is a complex file that contains a binary database stored in big endian format where the timestamps are stored in human readable (text type) [57]. The infor-

mation is stored in clear text with a specific plaintext format. This information is stored in relative byte address using pointers to reference this information where the position of the pointer is used as a base point address. The keychain uses the same schema design to store each database where each database contains different record schemas to store different type of information using its own record structures.

A keychain Mac OS X file starts with the database header follow by the database schema and the tables where each table stores a specific type of record or information. Finally, the file finishes with the required information to decrypt the protected information. The database header contains a magic value that identify a Mac OS X keychain, follow by the version of the keychain, the header size and the pointer called schema offset that indicates the byte offset in the file where the database schema starts. After the database header, the keychain has the database schema that contains the size of the schema, the number of tables and a list of pointers that indicates the relative address for each tables. The table address contended in the pointer is calculate as the database schema offset plus the value of the pointer. The number is the offset in byte where the table starts. Hence, knowing the number of tables and the offset to the table, the auditor is able to know where each table starts. Both records in database header and database schema are 4-byte fields in big endian.

Each table has the same table schema. The first record of the table is the size of the table, the second is the record type that the table stores follow by the number of records in the table and finally the offset of the first record. One table only contains one record type and all record type has the same size and structure. Then, knowing the record type the program knows the size of each record and knowing the number of record and where the first record starts, the program is able to read all the records of the table. The first record pointer is a relative pointer as a file offset in bytes that it is calculate as the address of the first byte of the table schema plus the value of the record. A diagram explained how Keychain works is showed in figure 6.11, where an hexadecimal structure with a real example are in figure 6.12 and 6.13.

Taking into account the role of a forensics purposes, the reversing process is done over the two most significant records: Record Internet Password scheme identified by the ID 0x80000001 and Record Generic Password scheme identify by 0x80000000. Both schemes store the credentials for email accounts, wireless networks, stored URLs, applications and secure notes. The structure of each record is explained in figures 6.14 and 6.15 where each field is a relative 4-byte pointer, where the offset address is calculate as the address of the record plus the value of the pointer minus one. Depending on the field, the information that the pointer points are different. It can be a string, a 4-byte text or a timestamp value. If it is a string it contains 4-byte with the number of character of the string follow by the characters whereas if it is a timestamp value the first 4-byte character is the year follow by five 2-byte characters that represent the month, day, hour, minute and second and finally ending by 2-byte of padding.

A parser to extract these two schemes from a keychain file has been implemented in the *keychains.py* parser as a proof of concept where the source code is available in the Appendix A. Furthermore, it has been implemented in Plaso Framework available since version 1.1. documented in Appendix B.

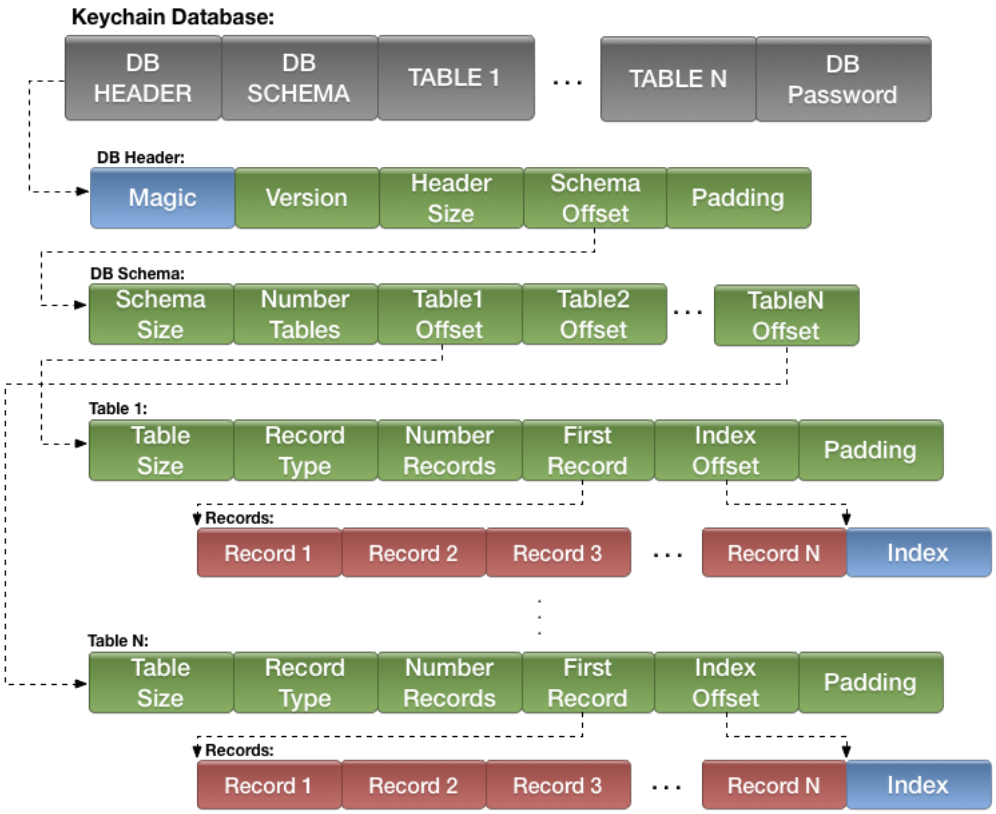

Figure 6.11: Keychain database structure.

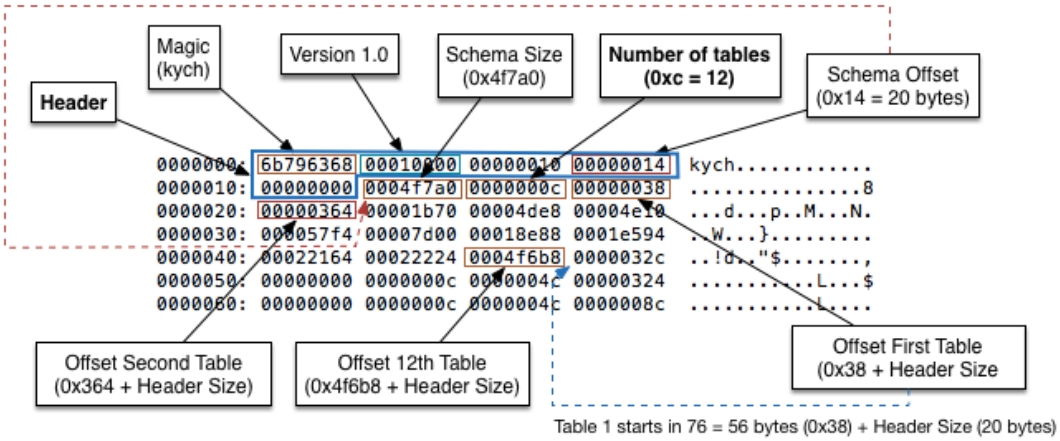

Figure 6.12: Keychain header and schema hexadecimal structure.

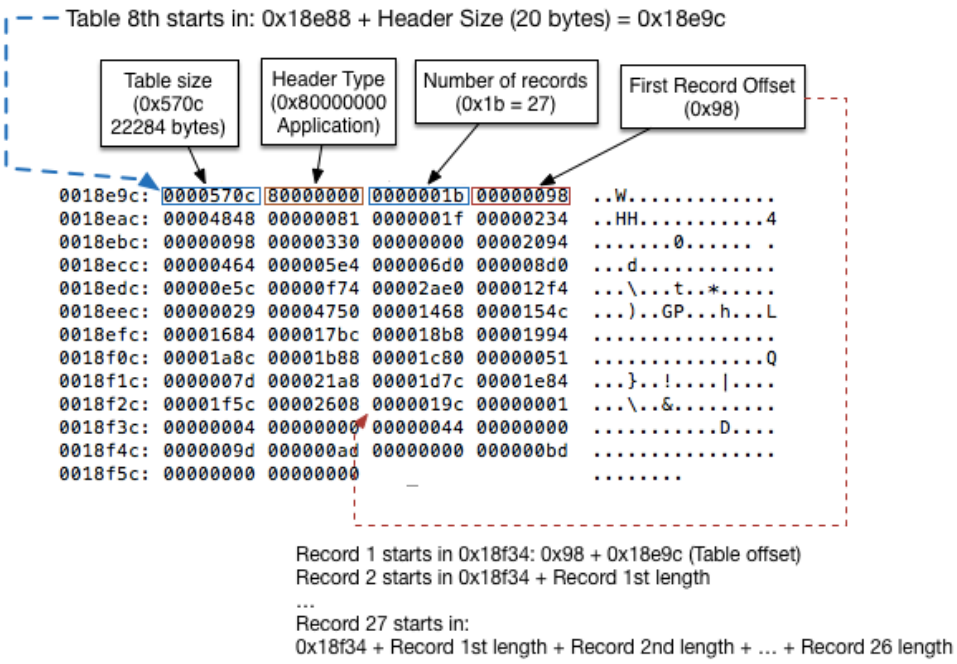

Figure 6.13: Keychain table header hexadecimal structure.

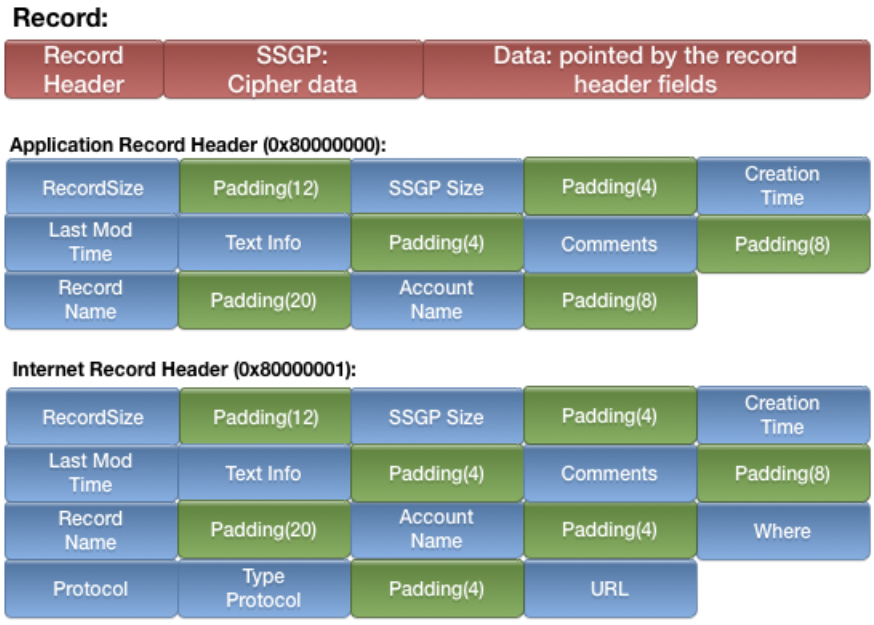

Figure 6.14: Keychain record structure.

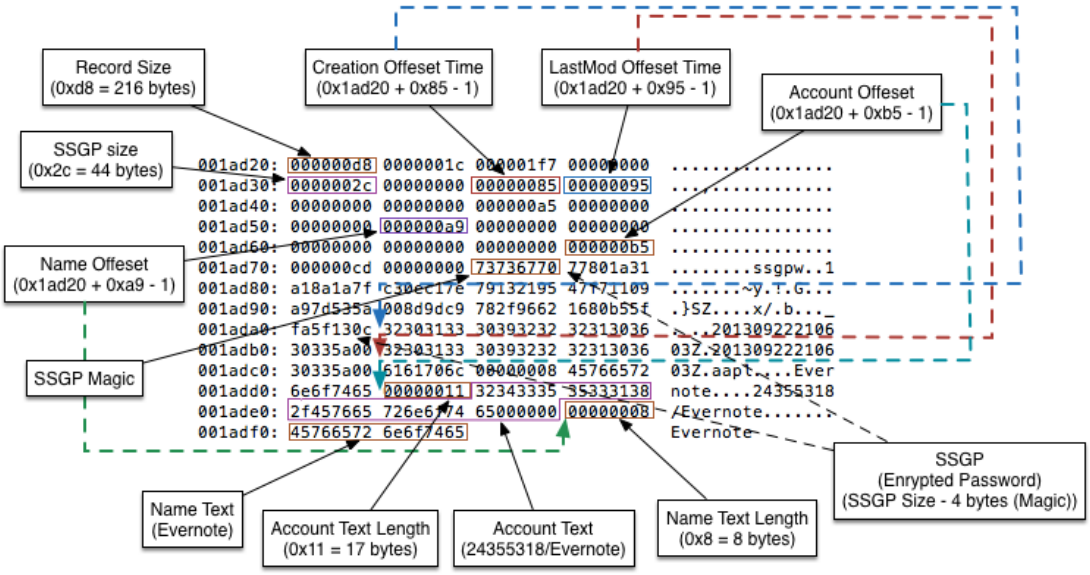

Figure 6.15: Keychain application record hexadecimal structure.

7 Case of study

This dissertation goals were to identity the persistent Mac OS X evidences, the type of information that they store, how this information can be extracted and provide an implementation that extract this information. Additionally, and not being the main target of the dissertation, a case of study has done using all the implemented plugins from the dissertation. The case of study is a proof of concept that shows how working with different evidences, the investigators can obtain relevant information to resolve the incident.

The environment for this case of study consisted in two VMWare virtual machines with a baseline Mac OS X 10.8 and 10.9 created using VMWare Fusion (figure 7.1). The host for both virtual machines was a hardening Linux. Additionally, a REMnux host was installed in the environment to hook the DNS queries and HTTP/HTTPS request. Over these Mac OS X virtual machines, some basic actions have been done to analyse how they modified the Mac OS X evidences identified during this dissertation. The process followed a cyclic process. First of all, creating a virtual machine snapshot, then executing the action that will be analysed, then extracting the evidences with Plaso over the virtual machine image, and finally, rolled back to the baseline status deleting the snapshot created before. To reduce the required time to analyse all the actions, Plaso was executed in a different Linux server provided by the university.

These main actions were divided in two groups: user interactions with the operation system and malicious software. The user interactions analysed in this case of study were:

- Mounting external partitions using CIFS services.
- Connect external USB devices and copying documents in the external devices.
- Connect the computer to the Internet using different interfaces and connecting the computer over a VPN tunnel.
- Acceding to some web page and storing the credentials in the browser.
- Trying to authenticate to the computer using the SSH service with invalid and valid credentials.

- Account activity: changing password, creating new users, calling the tool *sudo* with and without providing correct credentials.
- Common user behaviour searching for documents, open applications, removing files, etc.
- Printing documents.

Additionally, some malware for Mac OS X have installed to analyse the actions done over the operation system showed in table 7.1. Also, other external Mac OS X documentation has used to add more information to this case of study.

Name	**Type**	**SHA256**
Crisis	Java Trojan	4ef08344c7589cdc2b807cf5846d33a0 6a86609859bda114aa93f30f5f82dcb3
Careto (Mask A)	APT	a356f1650df2818c255a31bbc5313697 1aea17b4fdccd21b5bb3729c51359498
Leverage	Trojan/Backdoor	9bf2f2a273988a7e9b8318ae7a6aa26d 23ea8e5c841468605443c1a9a1fac193
CoinThief (C type)	Trojan	02caa8bb6b5180677cbe8a88b11f880c 5dc596ae92d6dc1cbefd7a30f23395fc

Table 7.1: Mac OS X Malware used.

Figure 7.1: Virtual machine running Mac OS X 10.9.

The process has identified different actions and patrons that are explaining as follow:

- **Application executed**: if the evidence comes from ASL (binary or plaint text) and the message contains *COMMAND=/bin/launchctl*. Also all the evidences coming from BSM where it has tokens process32, process64 and the extended version of both of them.
- **Application installed**: if the evidence comes from a property list and the key is InstallDate or the property list is from the history installation.
- **File downloaded**: if the evidence is a file which path contains *Download/'* or contains *'/Library/Mail/'* plus *Attachments* and the timestamp attribute is creation file (new file or crtime) [25].
- **Printed document**: any evidence coming from the control file CUPS IPP.
- **VPN**: BSM entry that contains a *socket_32, socket_128 or socket extended* token.
- **Malware**: different malware has been analysed obtaining the following set of rules:
  - Any ASL (binary or plaint text) log entry that contains *syspolicyd, unsafe-executable-type or iblaunch.dylib*.
  - A file which path contains *Trash/, .DS_Store.app or /private/var/db/BootCaches/* and the timestamps is creation file (new file or crtime).
  - If the Agents (LaunchAgents) launch property list is called:
    - com.google.softwareUpdateAgent.plist
    - jnana.plist
    - com.apple.UIServerLogin.plist
    - com.genieoinnovation.macetension.plist
    - checkvir.plist
    - com.apple.AudioService.plist [20]
    - UserEvent.System.plist
    - com.apple.launchport.plist [55]
- **User activity**: the configure Wi-Fi stored in *com.apple.airport.preferences.plist* and *wifi.log*, the configured bluetooth stored in *com.apple.Bluetooth.plist*, the searched terms stored in *com.apple.spotlight.plist*, the *InstallHistory.plist* that provided a history about application installed, the recent opened files using the *LSSharedFileList* library (recent files or bookmark attribute) stored in the directory *Users/user_name/Library/Preferences/*, the Document Versions database called *db.sqlite* and finally the keychains activity stored in *login.keychain and System.keychain* [25].
- **External devices**: any ASL (binary or plaint text) log entry that contains the string *USBMSC* or *fsevents* [44]. Additionally, the list of devices identified by the property list named *com.apple.sidebarlists.plist* (alias attributes).

- **Delete files**: if the evidence is a file which path contains */.Trash/.DS_Store* and the timestamp is creation time (new file or crtime) or modification time (metadata modification).
- **Account activity**: any ASL (binary or plaint text) log entry that contains any of these message text: *loginwindow, The authtok is incorrect, Establishing credentials, sudo, incorrect password attempts, create user, delete user, modify password* [89]. Also, any event coming from the UTMPX file which session ID is 0x7(USER_PROCESS) or 0x8 (DEAD_PROCESS). Additionally, the *passwordLastSetTime* system user property list contains the last time when the user password was changed. Finally, any BSM evidence that contains in the message the next strings *Type: session start, Type: OpenSSH login or session*.
- **Persistence:** this part identifies the software executed during the Mac OS X boot time [95].
  - List of well know hashes over the files */System/Library/CoreServices/boot.efi*, */mach_kernel* and */sbin/launchd*.
  - Malware detectors over the binary Mach-O kext file contained in */Library/Extensions/*/Contents/MacOS/**.
  - All the binaries identified by the property list in directories */System/Library/LaunchDaemons*, */Library/LaunchDaemons*, */System/Library/LaunchAgents*, */Library/LaunchAgents and $home/Library/LaunchAgents* where the property list attribute is *ProgramArguments* or *Program*.
  - Schedule task in directories */etc/crontab* and */usr/lib/cron/tabs*.
  - The Startup programs identified in the property list *StartupParameters.plist* in directories */System/Library/StartupItems/ and /Library/StartupItems/*.
  - Binaries added to the file */etc/rc.common or /etc/launchd.conf*.
  - Property list *com.apple.loginwindow.plist and com.apple.loginitems.plist* where the list of attributes (dictionary) contains the binary path to the executable.

As a result of the different test, an update set of rules to be used with *Plasm* (Plaso) has been implemented:

```
Application Execution
  data_type is 'macosx:application_usage'
  data_type is 'syslog:line' AND body contains 'COMMAND=/bin/launchctl'

Application Install
  data_type is 'plist:key' AND key is 'InstallDate'
  data_type is 'plist:key' AND plugin is 'plist_install_history'

AutoRun
  data_type is 'fs:stat' AND filename contains 'LaunchAgents/' AND
     timestamp_desc is 'crtime' AND filename contains '.plist'
```

```
  data_type is 'fs:stat' AND filename contains '/private/var/at/' AND
     timestamp_desc is 'crtime'

File Downloaded
  data_type is 'chrome:history:file_downloaded'
  timestamp_desc is 'File Downloaded'
  data_type is 'fs:stat' AND filename contains 'Download/' AND
     timestamp_desc is 'crtime'
  data_type is 'fs:stat' AND filename contains '/Library/Mail/' AND
     filename contains "Attachments" AND timestamp_desc is 'crtime'

Document Printed
  (data_type is 'metadata:hachoir' OR data_type is 'metadata:OLECF') AND
     timestamp_desc contains 'Printed'
  data_type is 'cups:ipp:event'

Account Activity
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'loginwindow'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'The authtok is incorrect'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'Establishing credentials'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'sudo'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'incorrect password attempts'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'create user'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'delete user'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'modify password'
  data_Type is 'mac:utmpx:event' AND (status contains 'USER_PROCESS' OR
     status contains 'DEAD_PROCESS')
  data_type is 'plist:key' AND plugin is 'plist_macuser' AND key is
     'passwordLastSetTime'
  data_type is 'mac:bsm:event' AND message contains 'Type: session start'
  data_type is 'mac:bsm:event' AND message contains 'Type: OpenSSH login'
  data_type is 'mac:asl:event' AND sender contains 'session'

External Devices
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'USBMSC'
  (data_type is 'syslog:line' OR data_type is 'mac:asl:event') AND body
     contains 'fsevents'

Delete Files
  data_type is 'fs:stat' AND filename contains '/.Trash/.DS_Store' AND
```

```
    (timestamp_desc is 'mtime' OR timestamp_desc is 'crtime')

Malware Activity
  data_type is 'mac:asl:event' AND sender is 'syspolicyd'
  data_type is 'mac:asl:event' AND body contains 'unsafe-executable-type'
  data_type is 'mac:asl:event' AND body contains 'liblaunch.dylib'
  data_type is 'fs:stat' AND filename contains '.Trash/' AND
     timestamp_desc is 'crtime'
  data_type is 'fs:stat' AND filename contains
     '/private/var/db/BootCaches/' AND timestamp_desc is 'crtime'
```

Security is mostly a superstition.
It does not exist in nature, nor do the children of men as a whole experience it.
Avoiding danger is no safer in the long run than outright exposure.
Life is either a daring adventure, or nothing.

Helen Keller

8 Conclusion

The number of new security threats in Mac OS X have been increasing during the last few years, especially those related with malware and intruders [54, 20]. However, the development of tools and the amount of research conducted to deal with these incidents have been minimal. Due to the widespread nature of computer forensics, this project has focused on the persistent evidences generated or managed by Mac OS X.

In the present document, the author has identified where the persistent Mac OS X evidences are stored, the process or mechanism used to store the evidences, the kind of information that is stored and the manner in which this information can be extracted. It was observed that a large number of these evidences were either not well documented or not documented at all. Furthermore, due to the implementation design decisions and backwards compatibility of Mac OS X, the evidences are stored following different criteria: little endian and big endian representation, text stored using *UTF-8, UTF-16 and ASCII*, different timestamp such as Epoch, HFS or Cocoa, relative file pointer address using 4-byte field and 8-byte field, and so on.

The most relevant evidences identified during the dissertation were: Apple System Log (6.1), Basic Security Module (6.2), Property list and their binary attributes (6.4, 5.4), UTMPX (6.3), system and users Keychains (6.8), CUPS IPP control files (6.6), the autologin functionality (5.3), Document version (6.7), persistence and boot time executables (5.6), Newsyslog rotate log tool (6.5), etc. The majority of these evidences are binary evidences unique to Mac OS X environment wherein a reversing process was required to be able to extract the information. Additionally, for each of this binary evidence, an explanatory diagram and hexadecimal real evidence has provided.

Finally, a parser for all of the documented evidences has implemented. During the first step, independent parsers has been developed to provide a full output information about how the information is extracted. These parsers are available on the Internet and are documented in Appendix A. Then, the forensics framework called Plaso has used to extending the tool to provide a Mac OS X support. The implementation over Plaso provides a solution that will be used by the forensics and incident response community in their daily jobs, where the work done during this year will be reflected in the visibility of the project and the university. The implemented parsers are available since *Plaso 1.1*

and documented in Appendix B.

With regards to future directions, there are different approaches that could be adopted in the coming years. The main ones among these are the following three:

First, persistent evidences improvement:

For each new version of Mac OS X the evidences must be checked to be sure that the format has not changed. Also, each new version of Mac OS X provided new functionalities that might store new evidences that must be analysed and implemented. Additionally, Mac OS X might have more evidences that were not identified during this document that should be study.

During the dissertation, the author has explained that some property lists are not only a list of attributes that contains basic values such as text, boolean or integer. Also, a few items contain binary attributes that have their own format like Alias or Bookmark (recent files) attributes. A research over all these binary attributes must be done. Also, instead of implement the extraction of these evidences with an external tool or inside plugins or parsers, the property list library [35] must be able to extract this information returning the output like if was another common attribute using a map python representation. Additionally, Apple has started to use the *DS_Store* files for not only to maintain information about how *Finder* shows the directories, maybe in the future new functionalities area added to these files and a new library should be required to extract the information that they contain.

Moreover, the dissertation has focus on the Mac OS X evidences related with the operation system; however, and due to time limits, the artifacts and logs coming from the independent applications such as Adium, iMail, iTunes or AppStore have not been analysed. These evidences must me identified, understand and implemented. A good starting point might be the Sean Cavanaugh spreadsheet [25] or Plaso implementation that already has support for some applications from other operation systems that are also used in Mac OS X environments such as Safari, Skype or Java IDX.

Second, persistent and volatile evidences integration:

During the background chapter, the different types of evidences that a forensic investigator can obtain were explained. Mainly, they are divided into two groups: persisted and volatile evidences. During the dissertation, the author focused only on the persistent evidences; despite both types of evidences provide useful information. An interesting approach will be unify both evidences and correlate the information, being able to obtain the answer about the incident. As an example, Plaso provides a plugin that add to the super timeline the output provided by Volatility framework (memory evidence) [41].

Third, correlation and behaviour analysis:

A behaviour analysis should be applied over the evidences obtaining the super timeline with all the evidences extracted, an automated tool must be able to analyse this output recognising specific patterns instead of working with an output file with thousands of actions done in the system. It can be considered as an

upper layer of abstraction in a manner of a group of actions are represented as an unique action that has a specific meaning. As an example of these upper actions might be mounting external devices, information leak, user authenticated, malware behaviour, security policy compliment, etc. In the *Chapter 7: Case of Study*, the author has done a research identifying basic behaviour and creating a set of rules that works with the Plaso fronted *Plasm*. However, this approach only permits to code basics rules based on where the evidences come from or text patrons without provide any machine learning algorithm or other artificial intelligence methods that permits complex and deeper rules.

A

Independent Parsers

Independent parsers that have been developed as a proof of concept. The parsers are stored in the URL *https://code.google.com/p/mac-osx-forensics*.

List of Parsers:
Alias Plist Parser
Binary Apple System Log parser
Basic Security Module
CUPS IPP control files parser
Autologin password
Mac OS X Keychains
Recent open files
Time Machine, Bluetooth, Wifi, Apple Acc, Updates, Spotlight plist parser
Mac OS X User property list evidence
UTMPX evidence
Wifi.log file log

Table A.1: Independent parsers.

B

Plaso implementation

Due to the number of lines implemented in Plaso, over ten thousand lines, the source code is not provided in the documentation. It can be obtained online from the source code of the project. It will be discussed below each of the implemented parser where for each parser it provides the URL for the parser (Parser), the unitary test file (Test), the output formatter (Formatter), and finally, an evidence example (Example) used by the unitary text to validate that the parser works as it is expected.

The project web page is *http://plaso.kiddaland.net/* and the source code is stored in *https://code.google.com/p/plaso/*. For formatting reason, the complete URL has omitted. To obtain the whole URL the *https://code.google.com/p/plaso/source/browse/* must be concatenated at the beginning to the URL provided in the following tables:

Evidence:	**Binary Apple System Log**.
Parser:	plaso/parsers/asl.py
Test:	plaso/parsers/asl_test.py
Formatter:	plaso/formatters/asl.py
Example:	test_data/applesystemlog.asl

Table B.1: Plaso ASL implementation.

Evidence:	**CUPS IPP control files**.
Parser:	plaso/parsers/cups_ipp.py
Test:	plaso/parsers/cups_ipp_test.p
Formatter:	plaso/formatters/cups_ipp.py
Example:	test_data/mac_cups_ipp

Table B.2: Plaso CUPS IPP implementation.

Evidence:	**Basic Security Module**.
Parser:	plaso/parsers/bsm.py
Test:	plaso/parsers/bsm_test.py
Formatter:	plaso/formatters/bsm.py
Auxiliar data:	plaso/unix/bsmtoken.py
Examples:	
	test_data/apple.bsm
	test_data/openbsm.bsm

Table B.3: Plaso BSM implementation.

Evidence:	**Application firewall log file**.
Parser:	plaso/parsers/mac_appfirewall.py
Test:	plaso/parsers/mac_appfirewall_test.py
Formatter:	plaso/formatters/mac_appfirewall.py
Example:	browse/test_data/appfirewall.log

Table B.4: Plaso Appfirewall.log implementation.

Evidence:	**Document versions**.
Parser:	plaso/parsers/sqlite_plugins/mac_document_versions.py
Test:	plaso/parsers/sqlite_plugins/mac_document_versions_test.py
Formatter:	plaso/formatters/mac_document_versions.py
Example:	test_data/document_versions.sql

Table B.5: Plaso document versions implementation.

Evidence:	**Mac OS X Keychains**.
Parser:	plaso/parsers/mac_keychain.py
Test:	plaso/parsers/mac_keychain_test.py
Formatter:	plaso/formatters/mac_keychain.py
Example:	test_data/login.keychain

Table B.6: Plaso Keychain implementation.

Evidence:	**Securityd log files**.
Parser:	plaso/parsers/mac_securityd.py
Test:	plaso/parsers/mac_securityd_test.py
Formatter:	plaso/formatters/mac_securityd.py
Example:	test_data/security.log

Table B.7: Plaso securityd.log implementation.

Evidence:	**Wifi.log plaintex file**.
Parser:	plaso/parsers/mac_wifi.py
Test:	plaso/parsers/mac_wifi_test.py
Formatter:	plaso/formatters/mac_wifi.py
Example:	test_data/wifi.log

Table B.8: Plaso wifi.log implementation.

Evidence:	**Syslog simple line using pyparsing**.
Parser:	plaso/parsers/syslog.py
Test:	plaso/parsers/syslog_test.py
Formatter:	plaso/formatters/syslog.py
Example:	test_data/syslog

Table B.9: Plaso BSD single line implementation.

Evidence:	**UTMPX**.
Parser:	plaso/parsers/utmpx.py
Test:	plaso/parsers/utmpx_test.py
Formatter:	plaso/formatters/utmpx.py
Example:	test_data/utmpx_mac

Table B.10: Plaso Mac OS X UTMPX implementation.

Evidence:	**Property list evidences**.
Parsers:	plaso/parsers/plist_plugins/macuser.py plaso/parsers/plist_plugins/softwareupdate.py plaso/parsers/plist_plugins/spotlight.py plaso/parsers/plist_plugins/spotlight_volume.py plaso/parsers/plist_plugins/timemachine.py
Test parsers:	plaso/parsers/plist_plugins/macuser_test.py plaso/parsers/plist_plugins/softwareupdate_test.py plaso/parsers/plist_plugins/spotlight_test.py plaso/parsers/plist_plugins/spotlight_volume_test.py plaso/parsers/plist_plugins/timemachine_test.py
Example:	test_data/InstallHistory.plist test_data/VolumeConfiguration.plist test_data/com.apple.SoftwareUpdate.plist test_data/com.apple.TimeMachine.plist test_data/com.apple.airport.preferences.plist test_data/com.apple.coreservices...ABC2.plist test_data/com.apple.spotlight.plist test_data/user.plist

Table B.11: Plaso Property list Mac OS X implementation.

C

Basic Security Module Tokens

The different Basic Security Module tokens implemented by the parsers are explained as follow:

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Token length
Unsigned Integer	8	Version
Unsigned Integer	16	Event type ID
Unsigned Integer	16	Modifier
Unsigned Integer	32	Epoch timestamp
Unsigned Integer	32	Microsecond

Table C.1: BSM Token Header 32.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Token length
Unsigned Integer	8	Version
Unsigned Integer	16	Event type ID
Unsigned Integer	16	Modifier
Unsigned Integer	64	Epoch timestamp
Unsigned Integer	64	Microsecond

Table C.2: BSM Token Header 64.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Token length
Unsigned Integer	8	Version
Unsigned Integer	16	Event type ID
Unsigned Integer	16	Modifier
Unsigned Integer	32	Net type
If Net type is 16:		
Unsigned Integer	128	IPv6 Address
Else:		
Unsigned Integer	32	IPv4 Address
Unsigned Integer	32	Epoch timestamp
Unsigned Integer	32	Microsecond

Table C.3: BSM Token Header 32 Extended.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Text length
Char	Text length * 8 bits	Text

Table C.4: BSM Token Text.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Text length
Char	Text length * 8 bits	Text

Table C.5: BSM Token Path.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	8	Status
Unsigned Integer	32	Return value

Table C.6: BSM Token Return 32.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	8	Status
Unsigned Integer	64	Return value

Table C.7: BSM Token Return 64.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Magic value
Unsigned Integer	32	Record length

Table C.8: BSM Token Trailer.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	8	Argument number
Unsigned Integer	32	-
Unsigned Integer	16	Text length
Char	Text length * 8 bits	Argument name

Table C.9: BSM Token Argument 32.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	8	Argument number
Unsigned Integer	64	-
Unsigned Integer	16	Text length
Char	Text length * 8 bits	Argument name

Table C.10: BSM Token Argument 64.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	32	Terminal port
Unsigned Integer	32	IPv4 Address

Table C.11: BSM Token Subject 32.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	32	Terminal port
Unsigned Integer	32	Net type
If Net type is 16:		
Unsigned Integer	128	IPv6 Address
Else:		
Unsigned Integer	32	IPv4 Address

Table C.12: BSM Token Subject 32 extended.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	64	Terminal port
Unsigned Integer	32	IPv4 Address

Table C.13: BSM Token Subject 64.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	32	Terminal port
Unsigned Integer	32	Terminal type
Unsigned Integer	32	Net type
If Net type is 16:		
Unsigned Integer	128	IPv6 Address
Else:		
Unsigned Integer	32	IPv4 Address

Table C.14: BSM Token Subject 64 extended.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Text length
Char	Text length * 8 bits	Text

Table C.15: BSM Token opaque (au_to_opaque).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Sequence value.

Table C.16: BSM Token Sequence (au_to_seq).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	IPv4 Address

Table C.17: BSM Token Address (au_to_in_addr).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Net type
Unsigned Integer	128	IPv6 Address

Table C.18: BSM Token Address Extended (au_to_in_addr_ext).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Char	160	Binary/Raw IPv4 Address

Table C.19: BSM Token IP (au_to_ip).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	8	Object type
Unsigned Integer	32	Object ID

Table C.20: BSM Token IPC (au_to_ipc).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Port number

Table C.21: BSM Token Port (au_to_iport).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Epoch timestamp
Unsigned Integer	32	microseconds
Unsigned Integer	16	Text length
Char	Text length * 8 bits	File name

Table C.22: BSM Token File (au_to_file).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	32	Terminal port
Unsigned Integer	32	IPv4 Address

Table C.23: BSM Token Process 32 (au_to_process32).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	64	Terminal port
Unsigned Integer	32	IPv4 Address

Table C.24: BSM Token Process 64 (au_to_process64).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	32	Terminal port
Unsigned Integer	32	Net type
If Net type is 16:		
Unsigned Integer	128	IPv6 Address
Else:		
Unsigned Integer	32	IPv4 Address

Table C.25: BSM Token Process 32 extended.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Audit UID
Unsigned Integer	32	Effective UID
Unsigned Integer	32	Effective GID
Unsigned Integer	32	Real UID
Unsigned Integer	32	Real GID
Unsigned Integer	32	PID
Unsigned Integer	32	Session ID
Unsigned Integer	64	Terminal port
Unsigned Integer	32	Net type
If Net type is 16:		
Unsigned Integer	128	IPv6 Address
Else:		
Unsigned Integer	32	IPv4 Address

Table C.26: BSM Token Process 64 extended.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Net type
Unsigned Integer	16	Port number
Unsigned Integer	32	IPv4 Address

Table C.27: BSM Token Socket 32.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Net type
Unsigned Integer	16	Port number
Unsigned Integer	128	IPv6 Address

Table C.28: BSM Token Socket 128.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Number of arguments

Table C.29: BSM Token Arguments.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Socket domain
Unsigned Integer	16	Socket type
If socket domain == 26:		
Unsigned Integer	16	IP type
Unsigned Integer	16	Source port
Unsigned Integer	128	Source IPv6 Address
Unsigned Integer	16	Destination port
Unsigned Integer	128	Destination IPv6 Address
Else:		
Unsigned Integer	16	IP type
Unsigned Integer	16	Source port
Unsigned Integer	32	Source IPv6 Address
Unsigned Integer	16	Destination port
Unsigned Integer	32	Destination IPv6 Address

Table C.30: BSM Token Socket Extended.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	8	How to print
Unsigned Integer	8	Data type
Unsigned Integer	8	Unit count

Table C.31: BSM Token Data (au_to_data).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	File mode
Unsigned Integer	32	UID
Unsigned Integer	32	GID
Unsigned Integer	32	File system ID
Unsigned Integer	64	File system node ID
Unsigned Integer	32	Device

Table C.32: BSM Token Attribute 32 (au_to_attr32).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	File mode
Unsigned Integer	32	UID
Unsigned Integer	32	GID
Unsigned Integer	32	File system ID
Unsigned Integer	64	File system node ID
Unsigned Integer	64	Device

Table C.33: BSM Token Attribute 64 (au_to_attr64).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	32	Status
Unsigned Integer	32	Return value

Table C.34: BSM Token Exit.

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Group number

Table C.35: BSM Token Group (au_to_newgroups).

Type	Length	Name
Unsigned Integer	8	BSM Token ID
Unsigned Integer	16	Text length
Char	Text length * 8 bits	Zonename text

Table C.36: BSM Token Zonename (au_to_zonename).

Bibliography

[1] Jerome Alet. Pkipplib library, 2006.

[2] Eugene Antsilevich. Capturing Timestamp Precision for Digital Forensics. *James Madison University Infosec Techreport*, page 36, 2009.

[3] Apple. Endutxent Library, 2006.

[4] Apple. CFimeutils Library, 2007.

[5] Apple. CRON BSD System manager's Manual, 2007.

[6] Apple. Launch Services Framework, 2007.

[7] Apple. Technical Note TN2083: Daemons and Agents, 2007.

[8] Apple. UTMPX Darwin C source code, 2009.

[9] Apple. Audit XNU source code, 2010.

[10] Apple. BSM XNU source code, 2010.

[11] Apple. OS X: Mac OS Extended format (HFS Plus) volume and file limits. *Apple Technical White Paper*, page 33, 2010.

[12] Apple. ASL Library, 2011.

[13] Apple. OS X Security. *Apple Technical White Paper*, page 13, 2012.

[14] Apple. Audit.log, Basic Security Module (BSM) file format, 2013.

[15] Apple. Date and Time Programming Guide, 2013.

[16] Apple. Kernel Extension Overview, 2013.

[17] Apple. OS X Lion: About Auto Save and Versions, 2013.

[18] Apple. Security Source Code, 2014.

[19] Apple. UTMP, WTMP, Lastlog, 2014.

[20] THe Mac Security Blog. New OS X Malware: Another Tibet Variant Found, 2013.

[21] Brian Carrier. *File System Forensic Analysis*. Addison Wesley, 2005.

[22] Harlan Carvey. *Windows Registry Forensics: Advanced Digital Forensic Analysis of the Windows Registry*. Syngress, 2011.

[23] Harlan Carvey. *Windows Forensic Analysis Toolkit*. Syngress, 2014.

[24] Harlan Carvey and Cory Altheide. *Digital Forensics with Open Source Tools*. Syngress, 2011.

[25] Sean Cavanaugh. OS X Lion Artifacts, 2011.

[26] CCL-ASL. Python Module for Parsing ASL files, 2013.

[27] Philip Craiger and Paul K. Burke. Mac forensics: Mac os x and the hfs+ file system. 2006.

[28] CUPS. CUPS Implementation of IPP, 2014.

[29] CyberMarshal. Mac Memory Reader, July 2012.

[30] Dan Doonan & Jacob Blend & Dan Doonan. Super Timeline. *The Senator Patrick Leahy Center for Digital Investigation, Champlain College*, page 15, 2013.

[31] Russell Shumway Dr. Eugene E. Schultz. *Incident Response: A Strategic Guide to Handling System and Network Security Breaches*. Sams, 2001.

[32] Sarah Edwards. Reading Mac BSM Audit Logs, 2012.

[33] Simon Key (EnCase). Mac OS X - Delving A Little. In *Mac OS X - Delving A Little*. CEIC Conferences, 2013, 2013.

[34] NetBSD Foundation. utmpx.h source code, 2008.

[35] The Python Software Foundation. Python Plistlib Library, 2014.

[36] Joaquin Moreno Garijo. Network Taps in NIDS, 2011.

[37] CERT Societe Generale. Incident Response Methodologies, 2012.

[38] Andrew Case Golden G. Richard3. In lieu of swap: Analyzing compressed ram in mac os x and linux. *Elsevier*, 2014.

[39] Kristinn Gudjonsson. Honing your log2timeline skills. In *Honing your log2timeline skills*. Digital Forensics Research Conference (DFRWS), 2013, 2013.

[40] Kristinn Gudjonsson. Reinventing the super timeline. In *Reinventing the super timeline*. NOLA, 2013, 2013.

[41] Kristinn Gudjonsson. Reinventing the super timeline. In *Reinventing the super timeline*. NSC, 2013, 2013.

[42] Kristinn Gudjonsson. Plaso Framework, 2014.

[43] Kristinn Gujónsson. Mastering the Super Timeline With log2timeline, 2010.

[44] Jason Hale. Automating USB Device Identification on Mac OS X, 2013.

[45] Andrew Hoog. *iPhone and iOS Forensics: Investigation, Analysis and Mobile Security for Apple iPhone, iPad and iOS Devices*. Syngress, 2007.

[46] Andrew Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Syngress, 2008.

[47] Charles Hornat. Malware Analysis, 2007.

[48] Charles Hornat. Volafox: Mac OS X & BSD Memory Analysis Toolkit, 2012.

[49] Apple Inc. Technical Note TN1150: HFS Plus Volume Format. *Apple Computer*, page 50, 2004.

[50] Leighton Johnson. *Computer Incident Response and Forensics Team Management*. Syngress, 2014.

[51] Sherri Davidoff Jonathan Ham. *Network Forensics: Tracking Hackers Through Cyberspace*. Prentice Hall, 2012.

[52] Rob Joyce. Mac Forensic Tools Using The Sleuth Kit. In *Mac Forensic Tools Using The Sleuth Kit*. ATC-NY, 2010, 2010.

[53] B. Kaliski. RFC2898: Password-Based Cryptography Specification, 2000.

[54] kaspersky. Flashback/Flashfake Malware, 2012.

[55] kaspersky. Unveiling "Careto" - The Masked APT, 2014.

[56] Simon Key. Examining Mac OS X User & System Keychains, 2013.

[57] Hyungjoon Koo Kyeongsik Lee. Keychain Analysis with Mac OS X Memory Forensics. *Korea University, CIST, DCWTC*, page 16, 2013.

[58] Jonathan Levin. *Mac OS X and iOS Internals To the Apple's Core*. John Wiley & Sons, Inc., 2013.

[59] Lucy. Inside the Mac OS X Kernel. In *Inside the Mac OS X Kernel*. Chaos Communication Congress, 2007.

[60] Cameron H. Malin. *Malware Forensics Field Guide for Linux Systems: Digital Forensics Field Guides*. Syngress, 2013.

[61] Mandiant. Memoryze for the Mac, 2012.

[62] Joachim Metz. Windowless Shadow Snapshots, 2012.

[63] Joachim Metz. Microsoft Internet Explorer (MSIE) Cache File (index.dat) files, 2013.

[64] Joachim Metz. Libfvde: FileVault2 Library, 2014.

[65] Omar Choudary & Felix Gröber & Joachim Metz. Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption. *University of Cambridge*, page 14, 2013.

[66] NetMarketShare. Desktop Operating System Market Share, 2012.

[67] Nickolay, trevorh, and Sheppy. Custom app bundles for Mac OS X, 2012.

[68] NTFS.com. HFS+, 2014.

[69] Oracle. SunSHIELD Basic Security Module Guide, 2010.

[70] Parki. Binary property list (plist) parser, 2013.

[71] Mila Parkour. OSX/Dockster.A, 2012.

[72] Hal Pomeranz. Solaris Basic Security Mode (BSM) Auditing, 2006.

[73] Robert. FSEventer: GUI interface for fslogger, 2010.

[74] RootedCon. RootedCon Security Conferences, 2014.

[75] R.Herriot S.Butler P.Moore R.Turner and J.Wenn. RFC2910: Internet Printing Protocol/1.1: Encoding and Transport, 2000.

[76] Paul Sanderson. A brief history of time stamps, 2010.

[77] SANS. Solaris C2 Auditing with BSM, 2001.

[78] SANS. Artifact analysis in SANS Digital Forensics and Incident Response, 2014.

[79] SANS. SANS Forensics Analyzer, 2014.

[80] Paul Cichonski & Tom Millar & Tim Grance & Karen Scarfone. Computer Security Incident Handling Guide 800-61 Revision 2. *National Institute of Standars and Technology*, page 79, 2014.

[81] Crucial Security. The Apple System Log, 2011.

[82] Jonathan Ham Sherri Davidoff. *Network Forensics: Tracking Hackers Through Cyberspace*. Prentice Hall, 2012.

[83] Michael Sikorski. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious* . No Starch Press, 2012.

[84] Amit Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley Professional, 2006.

[85] Amit Singh. A File System Change Logger, 2010.

[86] J.D. Smith. Timedog, view files saved by Time Machine, Jan 2009.

[87] 9To5 Staff. Top 10 Mac countries by market share, 2011.

[88] 9To5 Staff. Mac OS X source code plugins, 2014.

[89] Sud0man. Pac4Mac, 2013.

[90] Hashcat Team. Hashcat: advanced password recovery, 2014.

[91] Volatility team. Mac Memory Forensics, 2014.

[92] Chad Tilbury. New School Forensics. In *New School Forensics*. SANS Forensics, 2013.

[93] TrustedBSM. OpenBSM Project, 2012.

[94] Unknown. Dave Grohl: A password cracker for Mac OS X, 2014.

[95] Patrick Wardle. methods of malware persistence on mac os x. *Synack*, 2014.

[96] Robert N. M. Watson. The TrustedBSD Project, 2012.

[97] Lenny Zeltser. REMnux: A Linux Distribution for Reverse-Engineering Malware, 2013.

[98] Dino Dai Zovi. Advanced Mac OS X Rootkits. In *Advanced Mac OS X Rootkits*. Black Hat, 2009.