INTERNATIONAL
HELLENIC
UNIVERSITY

Memory Forensics and Bitcoin mining malware: Expanding the Volatility Framework for recovering Bitcoin keys and addresses from RAM acquired from multiple Operating Systems

Student Name: Dimotikalis Panagiotis

SID: 3301140003

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

DECEMBER 2015

THESSALONIKI – GREECE

Vasili
os
Katos

Digitally
signed by
Vasilios Katos
Date:
2016.05.05
19:22:39
+01'00'

INTERNATIONAL
HELLENIC
UNIVERSITY

Memory Forensics and Bitcoin mining malware: Expanding the Volatility Framework for recovering Bitcoin keys and addresses from RAM acquired from multiple Operating Systems

Student Name: Dimotikalis Panagiotis

SID: 3301140003

Supervisor:	Prof. Vasilis Katos
Supervising Committee Members:	Prof. Vasilis Katos
	Dr. Christos Berberidis

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

DECEMBER 2015

THESSALONIKI – GREECE

Abstract

Crime in the digital world has become a daily occurrence. Criminals adopt to new technologies with a faster pace than we are, people defending against new threats, giving them the advantage against unsuspecting victims. Their advantage is not due to their superiority; Offense has to succeed only once to be considered successful while defence has to succeed every single time to not be considered a failure.

Defending successfully against multiple threats using innovative technologies is hard and can only be achieved with careful planning and effective applying of knowledge acquired by examining those threats. Digital forensics is the epitome of this. Investigators need to have a firm grasp of up-to-date threats and how to locate and neutralize them. Memory forensics are the cornerstone of digital forensics. In recent years, memory acquisition and preservation of the state of a system when suspicious activity is undergoing, is the number one priority by every digital forensics investigator. To improve the capabilities of the investigator, in this thesis we examine the current threats associated with malware and the newly introduced technology of digital currencies, by proposing a series of enhancements to one of the most complete set of tools for memory analysis, the Volatility Framework.

Acknowledgments

I would like to thank my supervisor, Professor Vassilis Katos, for his advice throughout my thesis and his work as an information security researcher, inspiring me to become one. Zed Shaw for writing *Learn Python the Hard Way* book. Mikko Hypponen and Brod Aquilino of F-Secure, for providing malware samples for my research. Matt Suiche, for giving away for free a copy of MoonSols Memory Toolkit Pro. My parents for supporting me in every step of the way. Finally, I want to thank Emilie for believing in me, without you this would never have happened.

Dimotikalis Panagiotis

Date: 11/12/2015

Contents

1 Chapter 1 - Introduction

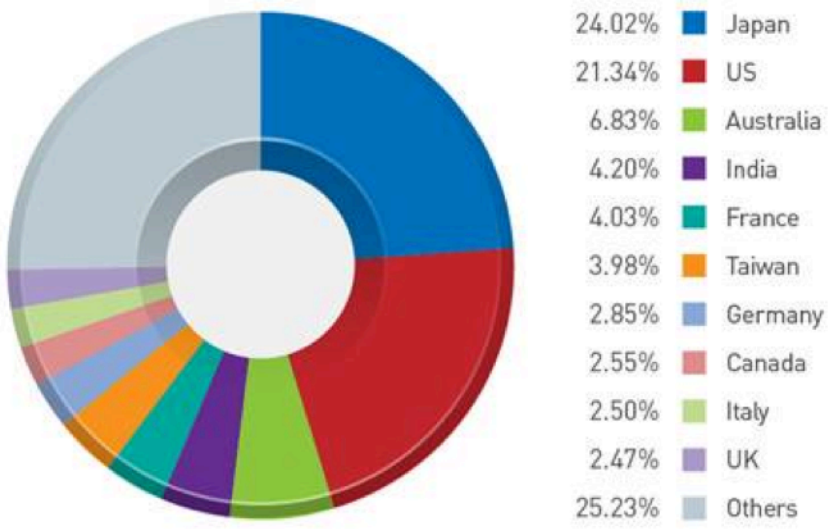
1.1 Background

This chapter will introduce the current trends in malware technologies, cybercrime economics and digital currencies, the fight against cybercrime and the important role of digital forensics and malware analysis in this struggle.

1.2 Problem Statement

Cybercrime is an ever-growing economy all over the world. Intel Corporation in 2014 estimated that the annual cost of cybercrime to the global economy is more than $400 billion [1], while other estimates claim losses of $575 billion or even a trillion US dollars annually [2][3]. To top this, many organizations and companies that are victims of such activity simply choose not to report cybercrimes to the authorities or the government, due to the nature of the crime and due to the fact that they do not know how this information is going to be used and interpreted by third parties; Doing so would most probably result in hurting their reputation and creating a notion of betrayal to their customers trust, thus leading to more economic damages. Since they have the option to keep the information to themselves and move on, they choose this solution in hope that they can survive the possible financial loss and resulting in the wide margin of the estimate losses mentioned earlier.

To make things worse, cybercriminals were introduced in their mischievous efforts to an unexpected ally; a solution to monetize their efforts in the most anonymous ways possible, digital currencies [4]. Although Bitcoin and other digital currencies were not invented in an effort to help and endorse such kind of behaviour, cyber crime is one of their top uses. Their high anonymity level and the almost non existing paper and digital trail they provide, resulted in cyber crime being one of their top uses. Moreover, automation and luck of need of human interaction for transactions involving digital currencies have resulted in the rise of software that tries to exploit this king of behaviour [5] [6]; Malware that occupies ones device usually without her knowledge in order to produce a digital currency or even worse, tries to steal digital currencies already residing on the device.



According to a report from Trend Micro [7], a security and antivirus company, in 2013 there was a rise of Bitcoin related malware in almost all major countries worldwide, with Japan and United States leading in infections rate as shown above. More recently sophisticated software known as ransomware, upon infection encrypts the files of the user and requests to be paid a bitcoin ransom within a matter of days in order for the user to gain back access to his encrypted files [8] [9].

Fortunately, security and antivirus companies like Trend Micro are doing their best in order to protect users, either via specialized software like antivirus and antimalware, either via sharing vital information on how to prevent and remove malicious software. However, the creators of said software, like in a cat and mouse game, are also trying to evolve their software in order to avoid being detected by antimalware solutions, making it really hard for victims to avoid paying [10] or both. This leads to rapid evolution of software from both perspectives, making it quite hard for an individual to keep up with all the techniques involved in fighting and creating advanced malware.

One can safely assume that sophisticated malware was not always the case. One of the first viruses, the Morris Worm [11] in November 1988, although exploiting vulnerabilities in Unix sendmail, finger and exec software, did not actually take advantage of its success, since -according to its creator- its only purpose was to figure out how many devices were connected to the Internet by then, it was easy to stop and its source code is a couple of hundred lines [12] only. It was In another case, according to a Microsoft report [13] one of the most well know cases of malware, the LoveLetter virus in 2000 relied on social engineering techniques in order to propagate itself further; the user had to manually open a mail attachment which was send to her, which led to code execution and further spreading of the virus. On the other hand, Code Red in July 2001 did not require any human interaction for spreading since it was exploiting a bug in Microsoft's software, Internet Information Services (IIS). Similarly, SQL Slummer [14] in 2003 relied on a buffer overflow vulnerability existing in Microsoft SQL Server products to propagate itself to almost every vulnerable server around the world within ten minutes upon its release, resulting to a slowdown of the whole Internet. In more recent years, in 2010 we witnessed one of the most complicated and extremely sophisticated malware that was ever discovered and probably created as of now; Stuxnet [15] [16]. Stuxnet managed from tens of thousands of kilometres away to infect its highly specific and well guarded targets, the computers of a nuclear facility in Iran that were never connected to the Internet, while managing to remain undetected for enough time to create malfunctions in the programmable logic controllers that were responsible for controlling the machinery of the facility, in a manner that was confusing enough in order to avoid detection by well trained engineers responsible for the well being of the faculty.

When examining a malicious software, in order to fully understand its impact, there are multiple perspectives that need to taken under consideration. There are the more obvious ones; the creator of the software or attacker, whose purpose is to infect as many victims as possible, spread the infection as far as possible, in the least required time. Then, it is the defender, who is dedicated in stopping the malicious intents of the attacker either by preventing the original infection or by blocking its further propagation. Finally, the third perspective is the examiner, referring to the one responsible for finding out how a particular malware works, documenting its behaviour and providing as many valuable information as possible for its neutralization. Usually the defender works in conjunction with the examiner or in some cases may be the same entity.

According to Kizza et all [17] “Computer forensics is the application of forensic science techniques to computer-based material. This involves the extraction, documentation, examination, preservation, analysis, evaluation, and interpretation of the computer-based material to provide relevant and valid information as evidence in civil, criminal, administrative, and other cases”. Although computer forensics ultimate target is the presentation of evidence in court, in a well documented and scientifically approved manner, the similarities of a computer forensic examiner’s ultimate goal compared to to the defender perspective that was mentioned earlier, are more than obvious: the defender acts as a forensic examiner would, with the only difference being that she –usually- will not have to present her findings in front of a judge. And as a computer forensic examiner, the defender has multiple areas of interest when dealing with a particular malware; from the connections it makes over the internet and the use of resources of the device it occupies to the files it uses to exploit the operating system and the RAM footprint it has while working in the background.

1.3 Research Question

In most of the aforementioned malware cases and generally every malware related case, careful examination of the infected device by specialized personnel will pinpoint the source of the problem, even in the case of extremely sophisticated malware like Stuxnet. This is a fact that the creators of the malicious software are aware of and try to counter-attack by making it harder and harder for investigators to analyse interpret their software and its behaviour. There are multiple ways to achieve this, one of which is to make the software have the smallest possible footprint both while operating and after it has fulfilled its purpose. A reasonable way to achieve such kind of tactics is for the software to fully operate in RAM and never touch the hard disk or any other parts of the device that might lead to persistent storage. By fully operating in RAM, apart from being faster, the malware will be erased by simply turning off the device, leaving no traces to be found.

The aim of this master thesis is to study and develop a way to examine traces of Bitcoin related malware that tries to implemented this scenario. More specifically, the creation of a plugin for an already acknowledged software in the computer forensics community, the Volatility Forensic Framework, will be introduced, in order to expand its capabilities for the newly appeared Bitcoin malware family. Moreover, the behaviour of various types of Bitcoin related malware will be examined, along with proposed ways of confronting them in multiple operating systems.

1.4 Methodology: Designing a Volatility Framework Plugin

In order to achieve a thorough understanding of how the Bitcoin malware family operates and be able to counterattack it, a detailed Literature Review will take place, regarding malware in general and in more details malware related to Bitcoin and other digital currencies. Techniques of malware creation, propagation and avoidance of detection will be examined and more specifically those that are related to RAM will be put under scrutiny. Related malware samples for all major operating systems will be acquired via multiple sources like security companies (F-Secure, Symantec, etc.) or sites dedicated to malware analysis and malware samples like virusshare.com, http://vxheaven.org and http://openmalware.org.

In addition, the functionality of Bitcoin, the technology behind it, the network infrastructure and the Bitcoin users operate, are going to be researched for us to be able to have a deep understanding of how a malware that tries monetize by taking advantage of Bitcoin. Alternative currencies and their structure will also be taken under consideration.

More over, the Volatility Forensic Framework will be thoroughly studied via its official guide written by its creators [18], books closely related to malware analysis [19] [20] and multiple presentations of other researchers in conferences around the world [21] [22] [23][24], in order to make sure that a deep understanding of its inner workings is at hand and a useful plugin can be created, following their guides. Plugins of the framework written by other users will be examined, while focusing specifically to those related to malware activity for every major operating system.

Multiple Test Beds will be created and used in order to test Bitcoin related malware to various conditions and operating systems, study their behaviour which will ultimately lead to valuable assumptions about their operation and ways to detect them. Linux and OSX will serve as hosts for multiple Virtual Machines running on top of them and having as guests at least two versions of each major operating system being in use today, with multiple snapshots for every one of them, in various stages of their infection and operations in order to collect as much information as possible regarding the behaviour of Bitcoin malware. Along with the Virtual Machines Snapshots, a number of memory dumps will be acquired which will be analysed using the Volatility Forensic Framework.

The proposed plugin will be heavily depended on the data acquired in the previous step. Relations and common patterns in the analysis of all the malware samples used will help the implementation. Python related literature [25] [26] will help overcome any difficulties that may arise.

Finally, the evaluation of the plugin will be tested using the Test Beds mentioned earlier. Weaknesses and strong points will be pointed out and suggestions for further development based on the experience acquired will be proposed.

1.5 Thesis outline

Each chapter of the dissertation and its contents will be outlined in order to provide the reader with a clear perspective and holistic view of the subject.

2 Chapter 2 – Literature Review

This chapter is dedicated to a thorough literature review related to the evolution of malware, digital currencies and more specifically Bitcoin, the Volatility Memory Forensics Framework and its plugins. An examination of the most significant books, research papers, major conference presentations and hands-on trainings was performed in order to gain an insight on the past and current trends of said technologies and their applications. The most significant findings are presented next.

2.1 Digital Currencies

In 2008 a paper written by Satoshi Nakamoto [27], a alias of an as of yet unknown person, introduced the idea of a peer-to-peer electronic currency that would allow transactions from one party to another without the need of a 3rd party financial institution taking verifying said transaction. However, this introduces a problem; Without the existence of a trusted 3rd party one or both ends of the transaction parties may double spend their funds. Nakamoto's proposed solution is the hashing of all transaction that are time stamped and verified by the network and then saved in an ongoing hash-based proof-of-work record that cannot be alternated. This electronic currency was named *Bitcoin* and was released as an open source software less than a year after the paper was published[28].

2.1.1 Definition and functionality

Bitcoin is the cornerstone of a series of technologies that are encapsulated into it. Its foundation is pure mathematics. Cryptography and computer networks are of great usage for Bitcoin. To make it clear of how Bitcoin works we will introduce an example of two users, Alice and Bob who would like to make a transaction.

Alice wants to transfer some Bitcoins to Bob, either because she bought something from him or she owns him some money. The main condition is that Alice has to make sure that Bob can accept Bitcoins, meaning that he has the appropriate means to do so. What Alice does when she sends Bob some Bitcoins, is sending him a series of numbers that have some certain mathematical attributes that make them hard to track, considering that there may be a malicious user, Eve, that wants to defraud the system in a number of ways, like as mentioned previously, double spending. This transaction like every other transaction is verified by the Bitcoin network, creating a permanent record binding it to the digital signature of Alice, a unique attribute given to every Bitcoin user in the network. For Alice to achieve the transaction she either has to use a special software called a *Bitcoin client* or use a service that can deal with the mechanics of the transaction (i.e. produce those numbers Alice will send) for her. From Bob's perspective, he too has to meet the same conditions; He has to have a *Bitcoin client* or use a service that will deal with Bitcoin transactions on his behalf.

One might wonder what Bob can do with the Bitcoins he receives from Alice. The obvious answer is that he can use them in order to acquire products or services from users or companies that they also accept Bitcoins. The other option is to exchange them for a physical currency of his choice via Bitcoin exchange services[1][2][3]. The value of a Bitcoin fluctuates in any given time, with the current rate being at 389 US Dollars at the time this lines were written. As stated earlier, Bitcoin transactions have some unique attributes when compared to transactions made with physical currency and these attributes may be the reason someone would want to use Bitcoin[29].

- Absence of 3rd party financial institution or services. It can lead to an increase of cost per transaction and add time consuming activities
- Privacy. Bitcoin transactions are pseudo anonymous[4] and private. Users making a transaction are certain that no other party can verify reliably that a transaction has taken place amongst them.
- Open source. Everyone can engage in Bitcoins transactions without the need of verification from 3rd parties (i.e. a bank) or knowledge of specific technical details.

[1] https://www.coinbase.com/
[2] https://www.kraken.com/
[3] https://www.bitstamp.net/
[4] http://www.coindesk.com/how-anonymous-is-bitcoin/

- Bitcoin is decentralized. When Alice sends Bob a Bitcoin, the transactions does not go through a 3rd party. As a result, there is no entity that can control the money supply or can seize the assets of a user or even reverse a transaction that has already taken place.

While decentralization may seem to have some disadvantages due to the absence of financial institutions that provide the security and verification of every transaction, this functionality is provided via the hash-based proof of work record that is shared via the peer-to-peer Bitcoin network.

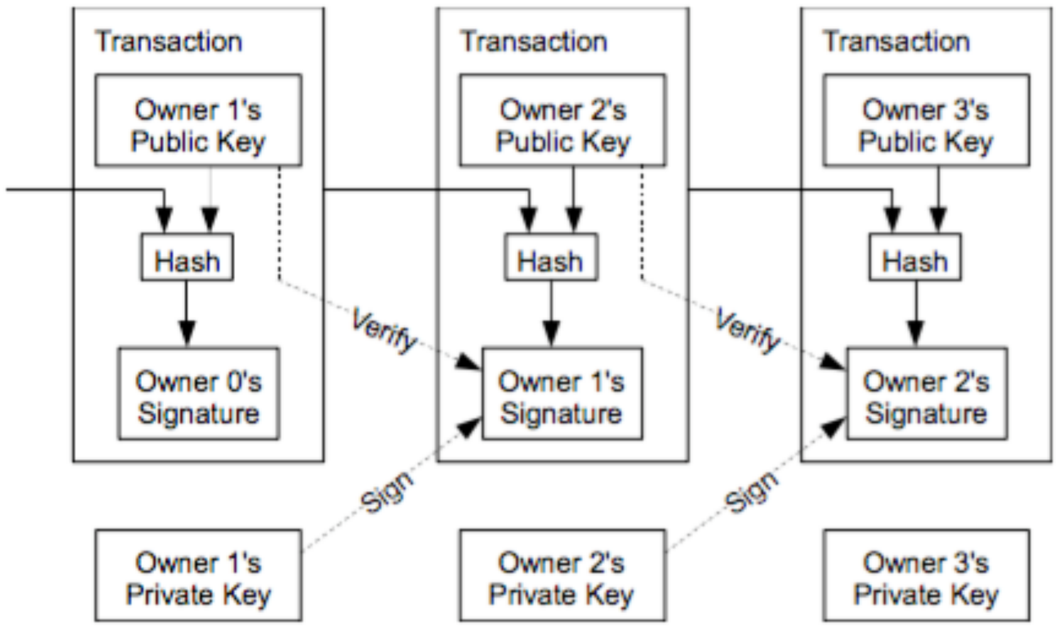


Figure 2-1, Bitcoin transactions verification schema [27]

One important part of the Bitcoin network and generally the whole ecosystem are the *miners.* Those are the users that participate in the network by providing computer power that is used in order to verify the unique attributes of every transaction that is being made and after validating that they are indeed legitimate, store the records of said transaction to the permanent record distributed to everyone which is called a *transaction block*. While validating transactions, miners will also add a record of their work, a *proof of work,* the verification of each transaction they achieved, which will result to a reward for them for all the work they provided. Moreover, miners will add to the *transaction block* an encoding of the previous valid transaction so as to achieve a level of continuity for the whole record shared by everyone in the network.

2.1.2 Types of Digital Currencies

Since the introduction of Bitcoin in 2009, thanks to its open source nature and the endorsement of its developer for creating *forks*[5] of its source code a number of alternative digital currencies have emerged, all based on the same principles.

- Litecoin[6]. A digital currency similar to Bitcoin developed in 2011 by Charles Lee, a former Google engineer [30]. The major difference from Bitcoin in the hashing algorithm implemented for the verification of the transactions. The exchange rate at the time this lines were written was 3.3 US Dollars to 1 Litecoin
- Darkcoin[7]. Created in 2013, its main focus is anonymity. As mentioned earlier, Bitcoin is pseudo anonymous and Darkcoin by mixing up transaction records and not providing the continuity used in Bitcoin, makes it harder to track the parties involved[31]. The exchange rate at the time these lines were written was 2.44 US Dollars to 1 Darkcoin.
- Dogecoin[8]. Introduced in 2013 and according to its creator Billy Markus its main purpose being the introduction of a digital currency with greater demographic that Bitcoin. As a result, while also using the same principles of Bitcoin, contrary to its predecessor and the cap of total Bitcoins that will ever be produced set to twenty-one million, Dogecoin has already more than 100 billion coins into circulation with more than 5 billion added each year. Due to this, the exchange rate at the time this lines were written was 0.13 US Dollars to 1000 Dogecoins.

[5] https://help.github.com/articles/fork-a-repo/
[6] https://litecoin.org/
[7] https://bitcointalk.org/index.php?topic=421615.0
[8] http://dogesilo.dogecanabank.info/

Those are some of the well know digital currencies existing. There is a vast majority of them, introducing slight changes to the core functions of Bitcoin. As of mid 2015 there is not a digital currency with deviations to the principles introduced by Satoshi Nakamoto in his original paper[27] [32].

2.1.3 Present Status

Since 2009 and its introduction, Bitcoin has an increasing popularity, growing bigger by the day. Started by basically having no value in 2009 and an exchange rate of one US cent per Bitcoin in 2010, it has come to have an average exchange rate of more that 300 US dollars in 2015.

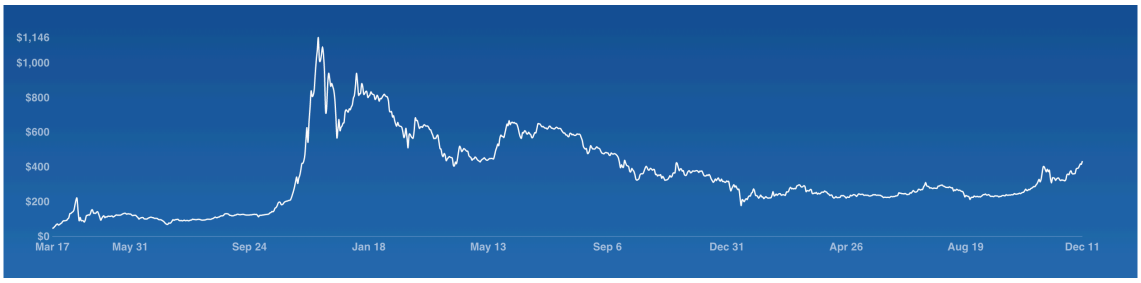


Figure 2-2, Bitcoin exchange rate in US dollars from March 2015 to December 2015

Major companies like Dell[9] and services like Paypal[10] have adopted Bitcoin as an alternative payment method for its tuition fees[11], the University of Nicosia is accepting Bitcoin as a payment method, while *blockchain* technology is being investigated as a foundation for financial transactions all over the world[32].

However, like every other technology, Bitcoin due to privacy attributes has risen to be the preferred way of payment in cyber crimes. From sites selling illegal substances[33], money laundering[34] to botnets used for illegal mining[5][7] and malware that encrypts the contents of hard drivers demanding ransom for their decryption[9], Bitcoin is always present.

[9] www.dell.com/bitcoin
[10]https://www.cryptocoinsnews.com/breaking-paypal-merchants-can-now-accept-bitcoin/
[11] http://www.unic.ac.cy/digitalcurrency

2.2 Bitcoin Malware

Gaining attraction has always its downsides and Bitcoin is not an exception to this rule. Since early 2013, when for the first time its exchange rate value was equal to a couple hundreds of US dollars, criminal activity targeting the Bitcoin ecosystem evolved and started targeting users of the Bitcoin network[6]. As users along with the vendors of operating systems and security software adapted to this new status quo by implementing a series of antimalware techniques and by making available information regarding malicious activity related to Bitcoin and how to avoid it, malicious users evolved too; By late 2013 new techniques implementing Bitcoin related malware emerged[9].

2.2.1 Definition

Oxford dictionary defines malware as a "software which is specifically designed to disrupt or damage a computer system"[12]. Malware related to Bitcoin clearly falls into this category of malicious software. Although there are not any known cases of computer systems that were damaged due infections from malware related to Bitcoin, the disruption of usage is the epitome of any related malware. Since 2013 where the price of Bitcoin started to rise (see earlier in this chapter[13]) criminal activity related to Bitcoin is on the rise and can be divided into two main categories; Illegal mining and *ransomware.*

[12] https://www.oxforddictionaries.com/definition/english/malware
[13] Chapter 2, *Present Status*

2.2.2 Types of Malware

Mining was the first practice of the Bitcoin ecosystem targeted by malicious users. Usually the unsuspected victim would install a software that came with bundled with miner. The miner would replicate itself into the system, register itself as a service or set a flag to start on every boot of the system and would mine for Bitcoins. Periodically would connect to a dedicated command server set by the developer of the malware, report or send the results of his mining activity and fetch new data for further mining. This would go on as long as the user did not notice its presence or the remote server was shut down.

There are many variations of this type of activity. For example, the miner is set to occupy only a maximum percent of the resources of the system he resides, in order to avoid detection for longer periods. The miner can also be setup to update itself daily via a web site or a server so as to avoid detection of an updated security software or a patched operating system.

Bitcoin mining on home computers is no longer a profitable activity even if the system is a state-of-the-art computer. Bitcoin mining has moved on to using dedicated *application specific intergraded circuits* (ASICs)[14], that exceed by far the power of any computer system a typical user might operate. Even in the case of a large botnet of thousands of bots, Bitcoin mining might not be the top priority for a malicious user considering the *value-for-money* ratio he can achieve.

However, even though it is well known that illegal home computer mining isn't worth the risk, there are still stories of malware related to it, even by legitimate companies who fall victims of rogue employees bundling miners into legitimate software[35].

Since mining is no longer the case for criminals, a different scheme related to Bitcoin has emerged in early 2014; *Ransomware.* A type of malware that encrypts a set of specific files or in some cases the whole hard drive and afterwards warns the user that she has to pay a ransom in Bitcoins within a short period of time usually of about 72 hours, in order to receive the decryption key of her files[9] [36].

These two kinds of Bitcoin malware are mainly targeting Microsoft Windows operating systems. Despite the fact that Bitcoin clients and Bitcoin miners are available for all three major operating system platforms (Windows, Linux, OS X), the overwhelming majority

[14]https://en.bitcoin.it/wiki/ASIC

of malware is developed for Microsoft's platform. Although there are recent cases of miners targeting Linux and OS X or the even more rare *ransomware* developed for these two operating systems, the ratio is disproportionally higher on behalf of Windows.

2.2.3 Evolution and Present Status

Unfortunately *ransomware* malware infections that mostly rely on *social engineering*[15] has proven to be lucrative for the its developers, resulting in more complicated attacks and updated versions of said malware; According to a research paper by Symantec in 2012 more that 5 million US dollars were extorted by the victims of such kind of malware in the United States alone[37]. As of late 2015 due to the efficient use of cryptography by the criminals, *ransomware* is still one of the most dangerous malware infections.

On the other hand, mining since it is no longer a lucrative business model for the criminals, has declined in volume, with the news stories like a miner for Linux focusing mostly on the fact that the miner was developed for a platform other than Windows.

[15]http://www.csoonline.com/article/2124681/leadership-management/security-awareness-social-engineering-the-basics.html

2.3 Memory Forensics

Memory acquisition has evolved to become the first step every forensic investigator and malware analyst makes when examining a system. When in a criminal investigation case failure to perform such a step may lead to the dismissal of the case[38]. Memory analysis is what follows the acquisition; The examination of the memory stored in a non-volatile manner, preserving evidence of activity present at the system at the time. A series of tools have been developed for all major operating systems and architectures in order to acquire and extract everything related to memory from a system. The analysis and interpretation of data residing on those files can be performed from a series of tools, one of which is the Volatility Framework, presented in Chapter 3.

2.3.1 Forensic Tools

This is a list of software tools that be used in order to acquire the memory of a system. The list in not complete and in no particular order, since unlike the state of hard disk acquisition software, there are no formal specifications that must be complied and thus be evaluated against them as a memory acquisition software developer. However there is a research paper regarding by S.Vömel and Stüttgen[39], introducing a platform for the evaluation of said software.

The main idea behind this type of memory acquisition is loading a module into the kernel which helps the software map the desired physical addresses of tasks running on the system into the virtual address space and then acquire the date from the virtual address space and store it to a storage device like a hard disk or a memory stick[18].

- Belkasoft Live RAM Caputer [16]. A free tool that according to the developer works in kernel-mode allowing it to bypass proactive anti-debugging protections.

[16] http://belkasoft.com/ram-capturer

- WindowsSCOPE Cyber Forensics[17], a memory acquisition suite that can also be used for analysis.
- mdd[18]. A memory acquisition tool for imaging Windows based computers officially supporting Windows versions up to Vista, but is also working on Windows 7.
- Memoryze[19]. Another free tool that supports capturing and analyzing memory. According to its developer it does not rely on API calls, allowing it to capture the full range of memory.
- KnTTools[20]. A tool for acquiring memory from Windows based systems.
- MoonSols Windows Memory Toolkit[21]. A toolkit containing all the utilities needed to perform any kind of memory acquisition or conversion during an incident response, or a forensic analysis for Windows platforms.
- FTK Imager[22]. A free tool supporting both x86 and x64 architectures of all major operating systems.
- Pmem Memory acquisition suite[23]. A series of tools for capturing memory from Windows (*WinPmem*) , Linux(*LinPmem*) and OS X (*OSXPmem*) by using a signed kernel module/driver in order to acquire the full range of memory of each system.
- LiME[24]. According to its developer, “a loadable kernel module (LKM) which allows for volatile memory acquisition from Linux and Linux-based devices, including Android”.
- Second Look Professional Edition[25]. A commercial product dedicated to memory acquisition from Linux based systems.
- fmem[26]. Another Linux kernel module, similar to LiME. It must be complied and loaded to the kernel of the system for memory acquisition. Its development has stoppen in 2010.
- Goldfish[27]. A memory acquisition tool over a FireWire connection, for OS X.

In case of virtualisation solutions, memory acquisition is usually provided by the developers. For example, Qemu and Xen allow the dumping of the contents of memory and VMware files can be analysed natively by the Volatility Framework.

[17] http://www.windowsscope.com/
[18] http://sourceforge.net/projects/mdd/
[19] https://www.fireeye.com/services/freeware/memoryze.html
[20] http://www.gmgsystemsinc.com/knttools/
[21] http://www.moonsols.com/windows-memory-toolkit/
[22] http://accessdata.com/product-download/?/support/adownloads#FTKImager
[23] http://www.rekall-forensic.com/docs/Tools/
[24] https://github.com/504ensicslabs/lime
[25] https://secondlookforensics.com/linux-incident-response/
[26] https://onyx.koli.ch/get/4480/fmem_current.tgz
[27] http://digitalfire.ucd.ie/?page_id=430

2.3.2 Importance of Memory Forensics

Throughout the history of digital forensics from early ‘80s to this day, there is a notable shift of thinking regarding the way a forensic investigator must operate when dealing with a crime scene involving a computer; One of the first steps he was required to take was the unplugging of the system from the network, in order to disrupt any effort of destroying digital evidence. This has changed; An investigator is now required to *preserve* the state of the system at all costs, in order to acquire as much evidence as possible.

This is a strong indicator that a great deal of system information in a volatile state and shutting it down or alternating its state like for example disconnecting it from the network may lead to loss of a huge portion of evidence.

Digital criminal activity has also shifted towards this direction; Advanced malware, like Stuxnet and Duqu[16][15], are operating in the memory of the system they infect. Performing a memory capture of a system ensures that the entire state of said system along with all the running applications including all the related data structures and variables is preserved in a non-volatile storage and can be investigated at will, without the eminent threat of alternating or loosing evidence.

Apart from malware that may only run in the memory of the system it infects, there is also the case of traditional malware that despite the fact that it resides on non-volatile storage it manages to hide its presence by injecting code to monitoring tools. While it may hide its presence it cannot hide its activity which at some point will rely on the memory of the system to achieve its goals. A memory capture while the malware operates will result in a number of indicators revealing its malicious purposes.

In other words, memory acquisition and memory analysis have come to be a crucial part of digital forensics and rightly so.

3 Chapter 3 - The Volatility Framework

In this chapter the Volatility Memory Forensics Framework is going to be analysed. A detailed overview of its features and structure is going to be presented in order for the reader to acquire an insight of its inner workings, its importance in the field of computer forensics and the ongoing development it undergoes. Moreover, notable plugins created by the open source community to expand its functionality are going to be presented. Finally, details about the test environment that is going to be used for the investigation of Bitcoin malware infections will be introduced along with software that accompanies this investigation.

3.1 Definition

The Volatility Framework is a complete collection of open source tools written in Python [40] [41] under the GNU General Public License for extracting and analysing digital artifacts from volatile memory (RAM) samples of multiple operating systems, acquired via 3rd party software. This extraction and analysis of the volatile memory is performed separately of the original system giving the investigator a great advantage considering that RAM constitutes a crucial and vital part of the runtime of a system [42].

3.2 Brief History and Overview

The Volatility Framework was firstly introduced in 2007, in one of the most important computer security conferences in the world that was taking pace in Washington DC at the time [43]. A. Walters and N. Petroni argued about the integral role of volatile memory analysis in digital forensics investigation process and the advantages it can provide to the forensics investigator. Moreover, they stressed the point of keeping multiple snapshots of

RAM and avoid performing a digital investigation on a live target when there are alternative solutions arguing that such practice will can alter the state of the device and thus become an obstacle to the investigation itself. To prove this, they created two virtual machines and monitored the state of their volatile memories for changes while they were idle. The results are shown in Table 3-1 that follows.

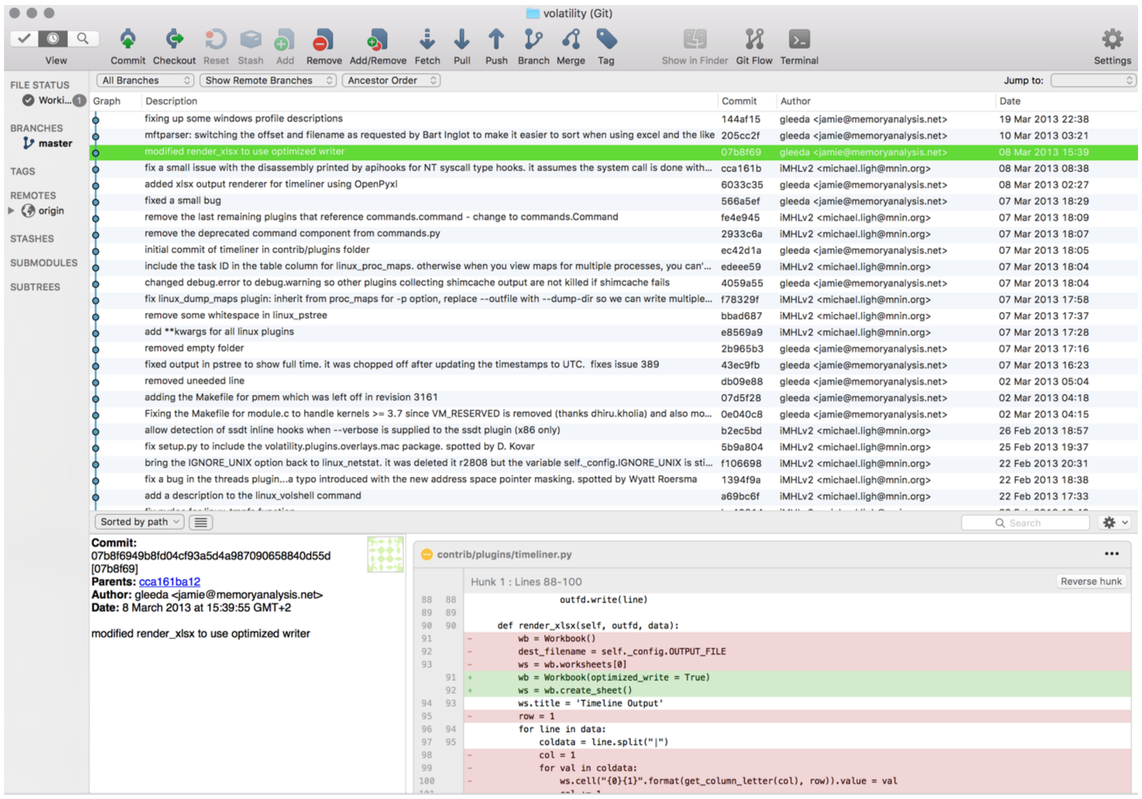
Table 3-1, Bytes of physical memory that changed as a function of time [43]

	256MB			512MB		
Minutes	Different	Total	% Same	Different	Total	% Same
0	0	268435456	100.00	0	536870912	100.00
5	14175676	268435456	94.7191	10059162	536870912	98.1263
10	16253700	268435456	93.9450	12272451	536870912	97.7141
60	25731960	268435456	90.4141	17804666	536870912	96.6836
120	54416944	268435456	79.7281	20542693	536870912	96.1736
900	70430438	268435456	73.7626	77252490	536870912	85.6106

Then, they proceeded in introducing Volatools, the software that will later become the Volatility Framework.

Up until their presentation every digital forensics investigation was focused on artifacts located on hard disks and other similar, permanent storage type devices. The volatile memory of the system was not taken under consideration although it almost certainly contains valuable information about the runtime of this very system and in some cases like the Stuxnet malware that were mentioned earlier in this thesis or the TrueCrypt encryption suite and its encryption keys, it contains the only artifacts that can lead to a successful forensic investigation.

The concept was welcomed by the digital forensics community and the software later to become the Volatility Framework, was embraced by developers and volunteers around the world due to its open source nature; It provided a cross-platform, modular and extensible platform that encouraged further work into a field that was making its first steps in the digital forensics era [41]. Since then, the Volatility Framework has evolved to one of the most highly used forensic tools and is being developed in a rapid pace with thousands of code commits. Its core developers have written a book dedicated solely to Volatility Framework [18], they perform trainings and have founded the Volatility Foundation, a non profit organization which was established to promote the usage of Volatility, protect its intellectual properties and help the advancement of memory forensics [41]



Description	Commit	Author	Date
fixing up some windows profile descriptions	144af15	gleeda <jamie@memoryanalysis.net>	19 Mar 2013 22:38
mftparser: switching the offset and filename as requested by Bart Inglot to make it easier to sort when using excel and the like	205cc2f	gleeda <jamie@memoryanalysis.net>	10 Mar 2013 03:21
modified render_xlsx to use optimized writer	07b8f69	gleeda <jamie@memoryanalysis.net>	08 Mar 2013 15:39
fix a small issue with the disassembly printed by apihooks for NT syscall type hooks. it assumes the system call is done with...	cca161b	iMHLv2 <michael.ligh@mnin.org>	08 Mar 2013 08:38
added xlsx output renderer for timeliner using OpenPyxl	6033c35	gleeda <jamie@memoryanalysis.net>	08 Mar 2013 02:27
fixed a small bug	566a5ef	gleeda <jamie@memoryanalysis.net>	07 Mar 2013 18:29
remove the last remaining plugins that reference commands.command - change to commands.Command	fe4e945	iMHLv2 <michael.ligh@mnin.org>	07 Mar 2013 18:09
remove the deprecated command component from commands.py	2933c6a	iMHLv2 <michael.ligh@mnin.org>	07 Mar 2013 18:07
initial commit of timeliner in contrib/plugins folder	ec42d1a	gleeda <jamie@memoryanalysis.net>	07 Mar 2013 18:05
include the task ID in the table column for linux_proc_maps. otherwise when you view maps for multiple processes, you can'...	edeee59	iMHLv2 <michael.ligh@mnin.org>	07 Mar 2013 18:04
changed debug.error to debug.warning so other plugins collecting shimcache output are not killed if shimcache fails	4059a55	gleeda <jamie@memoryanalysis.net>	07 Mar 2013 18:04
fix linux_dump_maps plugin: inherit from proc_maps for -p option, replace --outfile with --dump-dir so we can write multiple...	f78329f	iMHLv2 <michael.ligh@mnin.org>	07 Mar 2013 17:58
remove some whitespace in linux_pstree	bbad687	iMHLv2 <michael.ligh@mnin.org>	07 Mar 2013 17:37
add **kwargs for all linux plugins	e8569a9	iMHLv2 <michael.ligh@mnin.org>	07 Mar 2013 17:28
removed empty folder	2b965b3	gleeda <jamie@memoryanalysis.net>	07 Mar 2013 17:16
fixed output in pstree to show full time. it was chopped off after updating the timestamps to UTC. fixes issue 389	43ec9fb	gleeda <jamie@memoryanalysis.net>	07 Mar 2013 16:23
removed uneeded line	db09e88	gleeda <jamie@memoryanalysis.net>	02 Mar 2013 05:04
adding the Makefile for pmem which was left off in revision 3161	07d5f28	gleeda <jamie@memoryanalysis.net>	02 Mar 2013 04:18
Fixing the Makefile for module.c to handle kernels >= 3.7 since VM_RESERVED is removed (thanks dhiru.kholia) and also mo...	0e040c8	gleeda <jamie@memoryanalysis.net>	02 Mar 2013 04:15
allow detection of ssdt inline hooks when --verbose is supplied to the ssdt plugin (x86 only)	b2ec5bd	iMHLv2 <michael.ligh@mnin.org>	26 Feb 2013 18:57
fix setup.py to include the volatility.plugins.overlays.mac package. spotted by D. Kovar	5b9a804	iMHLv2 <michael.ligh@mnin.org>	25 Feb 2013 19:37
bring the IGNORE_UNIX option back to linux_netstat. it was deleted it r2808 but the variable self._config.IGNORE_UNIX is sti...	f106698	iMHLv2 <michael.ligh@mnin.org>	22 Feb 2013 20:31
fix a bug in the threads plugin...a typo introduced with the new address space pointer masking. spotted by Wyatt Roersma	1394f9a	iMHLv2 <michael.ligh@mnin.org>	22 Feb 2013 18:38
add a description to the linux_volshell command	a69bc6f	iMHLv2 <michael.ligh@mnin.org>	22 Feb 2013 17:33

Commit:
07b8f6949b8fd04cf93a5d4a987090658840d55d
[07b8f69]
Parents: cca161ba12
Author: gleeda <jamie@memoryanalysis.net>
Date: 8 March 2013 at 15:39:55 GMT+2

modified render_xlsx to use optimized writer

contrib/plugins/timeliner.py

Hunk 1 : Lines 88-100

Reverse hunk

```
                    outfd.write(line)

        def render_xlsx(self, outfd, data):
-           wb = Workbook()
-           dest_filename = self._config.OUTPUT_FILE
-           ws = wb.worksheets[0]
+           wb = Workbook(optimized_write = True)
+           ws = wb.create_sheet()
            ws.title = 'Timeline Output'
-           row = 1
            for line in data:
                coldata = line.split("|")
-               col = 1
-               for val in coldata:
-                   ws.cell("{0}{1}".format(get_column_letter(col), row)).value = val
```

Figure 3-1, Code commits as shown in github.com, a code repository and the ongoing development of the Volatility Framework

3.3 Structure and Functionality

The Volatility Framework in not the only memory forensics application. However, its features make it unique. Some of its core features according to its developers that makes it an ideal choice when it comes to memory forensics [18]:

- A single, cohesive framework. It can analyze memory snapshots originating from versions of all major operating systems of today like:
    - 64-bit Windows Server 2012 and 2012 R2
    - 32-bit and 64-bit Windows 8 and 8.1
    - 32-bit and 64-bit Windows 10
    - 32- and 64-bit Windows 7 (all service packs)
    - 32- and 64-bit Windows Server 2008 (all service packs)

    - 64-bit Windows Server 2008 R2 (all service packs)
    - 32- and 64-bit Windows Vista (all service packs)
    - 32- and 64-bit Windows Server 2003 (all service packs)
    - 32- and 64-bit Windows XP (SP2 and SP3)
    - 32- and 64-bit Linux kernels from 2.6.11 to 3.5
    - 32-bit 10.5.x Leopard (the only 64-bit 10.5 is Server, which isn't supported)
    - 32- and 64-bit 10.6.x Snow Leopard
    - 32- and 64-bit 10.7.x Lion
    - 64-bit 10.8.x Mountain Lion (there is no 32-bit version)
    - 64-bit 10.9.x Mavericks (there is no 32-bit version)

  along with all recent versions of Linux and in addition to memory snapshots of 32bit versions of Android, the mobile operating system of 80% of all mobile phones in the world [44].
- It is open source under the GNU General Public License v2, which means that it is completely free for everyone and gives access to its source code to every interested party.
- Python is the language used for its development. A well established programming language in the engineering and forensic world, easy to learn with huge potential while maintaining the easy-to-use mentality that it promotes.
- An extensible and scriptable application programming interface (API) giving the ability to developers to extend its capabilities in whichever way they may require: automatically explore kernel memory, create your own custom plugin or write a new malware sandbox.
- It is efficient. Volatility’s algorithms can analyze memory dumps from systems with 32 or 64 GB of RAM in a fraction of time while respecting the consumption of RAM of the system the analysis is performed.
- The community behind it apart from amateurs and individuals, has members coming from a law enforcement background like the Department of Justice of the United States, international companies like Google and Facebook, as well as the majority of antivirus and computer security firms.

The Volatility Framework supports multiple inputs for analysis [45]:

- Raw/Padded Physical Memory
- Firewire (IEEE 1394)
- Expert Witness (EWF)
- 32- and 64-bit Windows Crash Dump
- 32- and 64-bit Windows Hibernation
- 32- and 64-bit MachO files
- Virtualbox Core Dumps
- VMware Saved State (.vmss) and Snapshot (.vmsn)
- HPAK Format (FastDump)
- LiME (Linux Memory Extractor)
- QEMU VM memory dumps

Volatility is not a memory acquisition software. For this purpose, third party software should be used in order to acquire the memory dumps of a target device. There are multiple solutions for this purpose, while different operating systems require different software to achieve this. For Microsoft Windows there are solutions like MoonSols Windows Memory Toolkit [46] with free and retail versions, supporting 32bit and 64bit versions of Windows, compatible up to Windows 8.1.

For OSX there is OSXPmem [47], the Mac OS X Physical Memory acquisition tool which can capture memory dumps from versions 10.6 up to the most recent 10.11, while it primarily focused on 64bit versions of the operating system.

For Linux, depending on the version, there are also multiple solutions. The most common tool used is the Linux Memory Extractor (LiME) [48], a Loadable Kernel Module (LKM), which allows the acquisition of volatile memory from Linux and Linux-based devices. This last characteristic also makes it ideal for memory acquisition on Android devices, since Android is a Linux-based operating system. For older versions of Linux it is possible to capture the critical parts of memory by using the program dd [49] and coping the contents of the /dev/mem/ directory, a characteristic that has been deprecated in more recent versions.

As mentioned earlier, the Volatility Framework is implemented in Python. However, most of operating systems and the applications that run on them are primarily using the C programming language, with myriad uses of data structures in order to organize related variables and attributes [18]. In order for Volatility to translate all those C data structures in Python source files, it implements its own data structures, called *VTypes*. According to the authors of The Art of Memory Forensics, with VTypes one “can define structures whose member names, offsets, and types all match the ones used by the operating system one is analyzing, so that when she finds an instance of the structure in a memory dump, Volatility knows how to treat the underlying data (i.e., as an integer, string, or pointer).” [18][21]

This is an example of a C data structure and the equivalent VType for Volatility, written in Python:

```
struct process
    { int pid;
    int parent_pid;
    char name[10];
    char * command_line;
    void * ptv;
};
```

In this case, there are two integers, an array of characters, a pointer to a string and a void pointer. The translation of the above data structure to a VType data structure for use in Volatility Framework is as follows:

```
'process' : [ 26, {
    'pid' : [ 0, ['int']],
    'parent_pid' : [ 4, ['int']],
    'name' : [ 8, ['array', 10, ['char']]],
    'command_line' : [ 18, ['pointer', ['char']]],
    'ptv' : [ 22, ['pointer', ['void']]],
}]
```

Here, the name of the structure is declared (‘*process*’) which is the first dictionary key and it is accompanied by its size, which in this case is equal to 26. Then we proceed declaring the attributes of each member of the structure, their types and offsets. For example, the pointer to the string (‘*command line*’) is at offset 18 and its type is pointer.

This is done for all the data structures of an operating system which, as mentioned earlier, are myriads and usually change from version to version of a single operating system or even by its updates, let alone amongst different editions. Fortunately, there are ways to automate this process for most of the data structures existing in an operating system. For example, there is pdbparse [50] [51], a software that can be used to generate VTypes from the proprietary format of Microsoft's debugging symbols and thus be used in Volatility Framework. In other cases, when the data structures cannot be automatically produced, they must be manually developed after reverse engineering of specific parts of the operating system takes place.

Another important part of Volatility's inner workings are *overlays*. Referring to the data structure mentioned earlier[28], the use of void pointers is more than common in operating systems. Void pointers –in this case *ptv*- are used when the data type they point to is unknown at the time of the allocation. Overlays are used to fix up or patch automatically generated structure definitions when those type of pointers exist. The overlay that will be used in order to patch the ptv pointer when for example we know that it points to a process, has this structure[18]:

```
'process' : [ None, {
    'ptv' : [ None, ['pointer', ['process']]],
}
```

The 'None' attributes are used to point out that there should be no change regarding the offset and the size of the pointer, while 'void' becomes 'process' after the patch, indicate that the pointer from now on refers to a process.

A Volatility *object* is another crucial part of its core structure; According to the developers [18] [45], "it is an instance of a structure that exists at a specific address within an address space (AS)[29]. An *object class* gives the user the ability to extend the functionality of an object. In other words, she can attach methods or properties to an object that then become accessible to all instances of the object". As a result, plugins can share the same code, making their development a lot easier.

The combination of *VTypes*, *ovelays* and *object classes* that are derived from a specific operating system version including the hardware architecture, constitute the profile of this version of the operating system. Along with these characteristics, a profile includes the following:

[28] See page 14
[29] The total amount of memory that is allocated for a process

- Metadata: Information regarding the operating system's name, kernel version number, etc.
- System call information: Indexes and names of system calls
- Constant values: Global variables that can be found at hard-coded addresses in some operating systems
- Native types: Low-level types for native languages (usually C), including the sizes for all variable types
- System map: Addresses of critical global variables and functions (this information exists only in Linux and OSX profiles)

Profiles are necessary for correctly analyzing memory dumps and are already created for most of the versions and hardware architectures of all major operating systems. They can also be build and integrated into the framework manually.

One more major aspect of the Volatility framework are the plugins and the mechanisms they implement. The reason they are employed is the expansion of the existing capabilities of the framework in ways that differ from its original ones. For example, the support of newer CPU architectures can be achieved by an *address space*[30] plugin, whilst an analysis plugin is used to locate and analyze different components of an operating system, a malicious application, etc.

The basic plugin architecture of the latter according to the authors of The Art of Memory Forensics is a Python class that inherits from *commands* .Command which results in overriding some of the base methods like *calculate* and *render_text*. The first one and the code that it encapsulates is responsible for parsing the memory dump and analyzing any objects it locates, while *render_text* is used to pass the results of this analysis and present them on the terminal [18][41]. Following is a sample analysis plugin [52] for identifying the processes of a system which RAM has been acquired.

```
import volatility.utils as utils
import volatility.commands as commands
import volatility.win32.tasks as tasks
class ExamplePlugin(commands.Command):
  """This is an example plugin"""
  def calculate(self):
```

[30] An interface that provides flexible and consistent access to data in RAM, it handles the translation of virtual to physical address and provides the necessary background for interpreting differences in proprietary file formats

```
        """This method performs the work"""
        addr_space = utils.load_as(self._config)
        for proc in tasks.pslist(addr_space):
            yield proc
    def render_text(self, outfd, data):
        """This method formats output to the terminal.
        :param outfd | <file>
                data | <generator>
    """
    for proc in data:
        outfd.write("Process: {0}\n".format(proc.ImageFileName))
```

There are two possible ways of importing a plugin into the Volatility Framework and use it. Either copy the corresponding Python file into the plugins directory *(/volatility/plugins*) or direct Volatility to load the plugin by specifying its location via the - *-plugins* switch. In the first case, the framework will automatically load the plugin each time it is executed with no interference from the user, while on the latter case the user must point to the plugin every time. In both cases, multiple plugins can be used concurrently by simply adding them to the */plugins* directory or declaring them manually by adding colons after each declaration (*directory path to plugin1:directory path to plugin2:directory path to plugin3*). All available plugins that are picked up automatically by the framework and are loaded upon its execution can be listed by passing the *–info* switch to the main Python script.

Usage of the framework relies on the main Python script (vol.py) and switches that are passed to it via the command line. For example, in order to load a memory dump, one has to use the *–f* switch, followed by the directory that the file resides. In addition, the user must specify the operating system that the file was captured, the profile, that as mentioned earlier is a combination of *VTypes*, *ovelays* and *object classes*. If this information is not known, there is function that can be used prior to the analysis and determine the operating system. After this, additional options like plugin switches and arguments can be passed to the main python script for further analysis.

python vol.py –f <FILENAME> --profile=<PROFILE> <PLUGIN> [ARGS]

If the operating system profile information is not at the user's disposal, the *imageinfo* plugin is used to retrieve it. Following is an example of such usage:

python vol.py -f zeus.vmem imageinfo

The Volatility Framework will determine the most likely operating system that the file was created and present results similar to these:

```
Volatility Foundation Volatility Framework 2.5
INFO    : volatility.debug    : Determining profile based on KDBG search...
Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with
WinXPSP2x86)
AS Layer1 : IA32PagedMemoryPae (Kernel AS)
AS Layer2 : FileAddressSpace (/Users/Thesis/Memory Dumps/zeus.vmem)
PAE type : PAE
DTB : 0x319000L
KDBG : 0x80544ce0L
Number of Processors : 1
Image Type (Service Pack) : 2
KPCR for CPU 0 : 0xffdff000L
KUSER_SHARED_DATA : 0xffdf0000L
Image date and time : 2010-08-15 19:17:56 UTC+0000
Image local date and time : 2010-08-15 15:17:56 -0400
```

In this case, the capture of memory contents was performed on a Windows XP SP2 x86 system and was correctly identified by the framework as it can be seen by the suggested profiles option. From now on the analysis of this sample will be performed by passing the corresponding profile to the main python file of Volatility:

python vol.py -f zeus.vmem --profile=WinXPSP2x86 sockets

which results to the analysis shown below:

```
Volatility Foundation Volatility Framework 2.5
Offset(V)      PID   Port  Proto Protocol        Address         Create Time
---------- -------- ------ ------ --------------- --------------- -----------
0x80fd1008       4      0    47    GRE             0.0.0.0         2010-08-11 06:08:00 UTC+0000
0xff258008     688    500   17    UDP             0.0.0.0         2010-08-11 06:06:35 UTC+0000
0xff367008       4    445    6    TCP             0.0.0.0         2010-08-11 06:06:17 UTC+0000
0x80ffc128     936    135    6    TCP             0.0.0.0         2010-08-11 06:06:24 UTC+0000
0xff37cd28    1028   1058    6    TCP             0.0.0.0         2010-08-15 19:17:56 UTC+0000
0xff20c478     856  29220    6    TCP             0.0.0.0         2010-08-15 19:17:27 UTC+0000
0xff225b70     688      0  255    Reserved        0.0.0.0         2010-08-11 06:06:35 UTC+0000
0xff254008    1028    123   17    UDP             127.0.0.1       2010-08-15 19:17:56 UTC+0000
0x80fce930    1088   1025   17    UDP             0.0.0.0         2010-08-11 06:06:38 UTC+0000
0xff127d28     216   1026    6    TCP             127.0.0.1       2010-08-11 06:06:39 UTC+0000
0xff206a20    1148   1900   17    UDP             127.0.0.1       2010-08-15 19:17:56 UTC+0000
0xff1b8250     688   4500   17    UDP             0.0.0.0         2010-08-11 06:06:35 UTC+0000
0xff382e98       4   1033    6    TCP             0.0.0.0         2010-08-11 06:08:00 UTC+0000
```

```
0x80fbdc40       4      445   17    UDP             0.0.0.0            2010-08-11 06:06:17 UTC+0000
```

In this case the *sockets* argument was passed which prints a list of open sockets on the systems at the time of the capture. Another core plugin that can be used is *pstree* which prints the process list tree of the system, along with information about the PID[31], threads, handles and relationships:

```
Volatility Foundation Volatility Framework 2.5
Name                                             Pid  PPid   Thds   Hnds Time
-------------------------------------------------- ------ ------ ------ ------ ----
 0x810b1660:System                               4      0       58       379     1970-01-01 00:00:00
UTC+0000
. 0xff2ab020:smss.exe                          544      4        3        21      2010-08-11 06:06:21
UTC+0000
.. 0xff1ec978:winlogon.exe                     632    544      24      536      2010-08-11 06:06:23
UTC+0000
... 0xff255020:lsass.exe                       688    632       21      405      2010-08-11 06:06:24
UTC+0000
... 0xff247020:services.exe                    676    632      16      288      2010-08-11 06:06:24
UTC+0000
.... 0xff1b8b28:vmtoolsd.exe                  1668    676      5       225      010-08-11 06:06:35
UTC+0000
..... 0xff224020:cmd.exe                       124   1668      0       ------     2010-08-15 19:17:55
UTC+0000
.... 0x80ff88d8:svchost.exe                    856    676      29      336       2010-08-11 06:06:24
UTC+0000
.... 0xff1d7da0:spoolsv.exe                   1432    676      14      145       2010-08-11 06:06:26
UTC+0000
.... 0x80fbf910:svchost.exe                   1028    676     88     1424     2010-08-11 06:06:24
UTC+0000
..... 0x80f60da0:wuauclt.exe                  1732   1028      7       189       2010-08-11 06:07:44
UTC+0000
..... 0x80f94588:wuauclt.exe                   468   1028      4       142       2010-08-11 06:09:37
UTC+0000
..... 0xff364310:wscntfy.exe                   888   1028      1       40        2010-08-11 06:06:49
UTC+0000
.... 0xff217560:svchost.exe                    936    676     11      288       2010-08-11 06:06:24
UTC+0000
.... 0xff143b28:TPAutoConnSvc.e  1968    676      5       106       2010-08-11 06:06:39
UTC+0000
..... 0xff38b5f8:TPAutoConnect.e  1084   1968      1       68        2010-08-11 06:06:52
UTC+0000
.... 0xff22d558:svchost.exe                   1088    676      7        93       2010-08-11 06:06:25
```

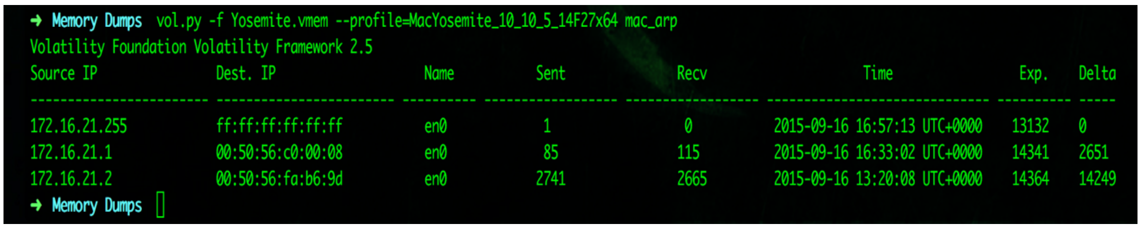
[31] Process Identifier, a unique number to identify an active process

```
UTC+0000
.... 0xff218230:vmacthlp.exe        844    676    1      37       2010-08-11 06:06:24
UTC+0000
.... 0xff25a7e0:alg.exe             216    676    8      120      2010-08-11 06:06:39
UTC+0000
.... 0xff203b80:svchost.exe         1148   676    15     217      2010-08-11 06:06:26
UTC+0000
.... 0xff1fdc88:VMUpgradeHelper     1788   676    5      112      2010-08-11 06:06:38
UTC+0000
.. 0xff1ecda0:csrss.exe             608    544    10     410      2010-08-11 06:06:23
UTC+0000
 0xff3865d0:explorer.exe            1724   1708   13     326      2010-08-11 06:09:29
UTC+0000
. 0xff374980:VMwareUser.exe         452    1724   8      207      2010-08-11 06:09:32
UTC+0000
. 0xff3667e8:VMwareTray.exe         432    1724   1      60       2010-08-11 06:09:31
UTC+0000
```

Similarly, Volatility can be used to analyze files acquired from Apple's Mac OS X family operating systems. In this case, there is an additional step one must make in order to properly configure the framework for such a task; the corresponding operating system profile must be added since it is not included by default. The Volatility Foundation maintains an active list of those profiles[32]. The user can download the profile she requires and paste the zip file to *volatility/plugins/overlays/mac* or *volatility/plugins/overlays/linux* directories, depending on the system the analysis refers to. In the example that follows, the OS X Yosemite 10.10.5 profile was used:

python vol.py -f Yosemite.vmem --profile=MacYosemite_10_10_5_14F27x64 mac_arp

The result can be shown in Figure 3-2 bellow. The plugin mac_arp was used in order to print the ARP table[33] of the system



```
➜ Memory Dumps  vol.py -f Yosemite.vmem --profile=MacYosemite_10_10_5_14F27x64 mac_arp
Volatility Foundation Volatility Framework 2.5
Source IP                Dest. IP                 Name       Sent              Recv              Time                           Exp.       Delta
------------------------ ------------------------ ---------- ----------------- ----------------- ------------------------------ ---------- -----
172.16.21.255            ff:ff:ff:ff:ff:ff        en0                1                 0         2015-09-16 16:57:13 UTC+0000   13132      0
172.16.21.1              00:50:56:c0:00:08        en0               85               115         2015-09-16 16:33:02 UTC+0000   14341      2651
172.16.21.2              00:50:56:fa:b6:9d        en0             2741              2665         2015-09-16 13:20:08 UTC+0000   14364      14249
➜ Memory Dumps
```

Figure 3-2, Printing the ARP table of an OS X system

[32] https://github.com/volatilityfoundation/profiles

[33] A cache of the *Address Resolution Protocol* used for storing IP addresses and their resolved Ethernet physical addresses.

Another example is *mac_dmesg* which implements a well know command for UNIX-like operating systems, *dmesg*

python vol.py –f Yosemite.vmem --profile=MacYosemite_10_10_5_14F27x64 mac_dmesg

which is used to print the message buffer of the kernel:

```
Volatility Foundation Volatility Framework 2.5
x: svga: Start: FB size=0x300000, FIFO size=0x200000
Apple16X50UARTSync1: Detected 16550AF/C/CF FIFO=16 MaxBaud=115200
gfx: svga: Start: host_bpp=32, bpp=32, num_displays=1
Apple16X50UARTSync2: Detected 16550AF/C/CF FIFO=16 MaxBaud=115200
gfx: fb: start: maxWidth 1920 maxHeight 1080 vramSize 134217728
gfx: fb: setDisplayMode: (1) wxh=1024x768, 32 4096
gfx: svga: SetMode: mode w,h=1024, 768 bpp=32
gfx: svga: SetMode: pitch=4096
gfx: fb: setDisplayMode: Display ID=1, Depth ID=0
gfx: fb: setDisplayMode: wxh=1024x768, bpp=32, pitch=4096
gfx: gfx: UpdateTraces: Enabling traces.

Previous shutdown cause: 0
Waiting for DSMOS...
0x1face000, 0x00000000  Intel82574L::setLinkStatus - not active
0x1face000, 0x00000000  Intel82574L::setLinkStatus - not active
0x1face000, 0x00000000  Intel82574L::setLinkStatus - not active
0x1face000, 0x00000000  Intel82574L::setLinkStatus - not active
0x1face000, 0x00000000  Intel82574L::setLinkStatus - not active
Ethernet [Intel82574L]: Link up on en0, 1-Gigabit, Full-duplex, No flow-control,
Debug [796d,ac08,01e1,0200,41e1,7c00]
0x1face000, 0x0000000a  Intel82574L::setLinkStatus - active
DSMOS has arrived
[snip]
```

In general, several of the command options supported by Volatility can used in conjunction with most of the plugins regardless of the file that is being analyzed. Included there is a help menu that can be displayed by passing the *–help* (or *–h*) argument:

```
Volatility Foundation Volatility Framework 2.5
Usage: Volatility - A memory forensics analysis platform.

Options:
  -h, --help          list all available options and their default values.
                      Default values may be set in the configuration file
                      (/etc/volatilityrc)
  --conf-file=/Users/Gi0/.volatilityrc
                      User based configuration file
  -d, --debug         Debug volatility
  --plugins=PLUGINS   Additional plugin directories to use (colon separated)
  --info              Print information about all registered objects
```

```
--cache-directory=/Users/Gi0/.cache/volatility
                      Directory where cache files are stored
--cache               Use caching
--tz=TZ               Sets the (Olson) timezone for displaying timestamps
                      using pytz (if installed) or tzset
-f FILENAME, --filename=FILENAME
                      Filename to use when opening an image
--profile=WinXPSP2x86
                      Name of the profile to load (use --info to see a list
                      of supported profiles)
-l LOCATION, --location=LOCATION
                      A URN location from which to load an address space
-w, --write           Enable write support
--dtb=DTB             DTB Address
--shift=SHIFT         Mac KASLR shift address
--output=text         Output in this format (support is module specific, see
                      the Module Output Options below)
--output-file=OUTPUT_FILE
                      Write output in this file
-v, --verbose         Verbose information
-g KDBG, --kdbg=KDBG  Specify a KDBG virtual address (Note: for 64-bit
                      Windows 8 and above this is the address of
                      KdCopyDataBlock)
--force               Force utilization of suspect profile
--cookie=COOKIE       Specify the address of nt!ObHeaderCookie (valid for
                      Windows 10 only)
-k KPCR, --kpcr=KPCR  Specify a specific KPCR address
```

Also in the help menu a list of plugins loaded by the framework and are available for use is show.

3.4 Existing Plugins

Plugins shown in the help menu of Volatility are a fraction of the whole list that is actually supported. This list of more than 250 plugins (see Appendix) for all major operating systems can be printed by passing the -- *info* argument:

python vol.py -- info

The functionality of those plugins cover a vast area of interest regarding forensic analysis. There are plugins for detect API[34] hooks in process and kernel memory, reconstructing the a browser’s cache memory and history, connecting to the file via a shell or even for

[34] Application Program Interface

saving a pseudo-screenshot of what was on the screen of the user at the time by using the Windows GDI system[35].

Apart from the core plugins included by default in the framework, there are more that forty plugins that are developed and maintained by the Volatility community and are available to everyone[36]. Moreover, since 2013[37] and every year the Volatility Foundation is holding a contest for the top three plugins produced by members of the community, awarding them with various ways. For example, in 2013 a OS X rootkit detection plugin was introduced by Cem Gurkok [53]–[55]. Thanks to his work, Volatility is able to detect

- Direct syscall table modification
- Syscall function inlining (ie DTrace hooks)
- Patching the syscall handler (ie, shadow sycall table)
- Hooked functions in kernel/kext symbol tables
- Modified IDT descriptors
- Modified IDT handlers

In 2014 he expanded the capabilities of his plugins by being able to identify and recover bitcoin keys and addresses (see Appendix), enumerating threads of tasks running on a Mac and detecting suspicious behavior regarding the TrustedBSD policy list [45]. Another useful feature of plugins is their ability to register their own options, which can be viewed by the --help or –h switches that can accompany a plugin. For example:

python vol.py pstree --help

will produce the following output:

```
Volatility Foundation Volatility Framework 2.5
Usage: Volatility - A memory forensics analysis platform.

Options:
 [snip]
  --force              Force utilization of suspect profile
  --cookie=COOKIE       Specify the address of nt!ObHeaderCookie (valid for
                        Windows 10 only)
  -k KPCR, --kpcr=KPCR  Specify a specific KPCR address
```

[35] Microsoft Windows Graphics Device Interface
[36] https://github.com/volatilityfoundation/community
[37] http://www.volatilityfoundation.org/#!2013/c19yz

```
Module Output Options: dot, greptext, html, json, sqlite, text, xlsx

---------------------------------
Module PSTree
---------------------------------
Print process list as a tree
```

Along with the plugins list printed to the terminal by the *--info* argument, a list of the profiles that are supported (see Appendix), including those loaded manually like the OS X profile shown in a previous example is presented. In addition to these profiles, there are profiles for both x86 and x64 architectures of six major distributions of Linux[38], including Fedora, RedHat, Debian and OpenSUSE and more than sixty profiles for OS X version 10.5 and onwards[39]

3.5 Plugin Case Studies

In this section a number of case studies will be presented, regarding the functionality of the framework on analysing files captured from every major operating system today.

3.5.1Windows

One of the most complicated and advanced malware that was ever created is Stuxnet[15][16]. Its origins are yet to be know, but due to its extremely sophisticated nature it is believed to be a creation of a state or even a coalition of states. Its purpose was to sabotage Iran's nuclear program by infecting PLCs[40] of a specific nuclear factory in Iran in order to introduce malfunctions in a way that they would not be traceable, not even by the engineers working at the premises. It was using four different zero-day exploits[56]

[38] https://github.com/volatilityfoundation/profiles/tree/master/Linux
[39] https://github.com/volatilityfoundation/profiles/tree/master/Mac
[40] Programmable logic controller

for Microsoft's Windows operating system and was designed in such elaborate way that it could reach its targets although they were not connected to the Internet.

Since its discovery in 2010, Stuxnet has been analyzed by multiple digital investigators and samples are available from multiple sources[41][42][43][44] for everyone interested to investigate further. In order to analyze the behavior of the malware with Volatility, a virtual machine using VMWare's Workstation Pro[45] software was created and later infected with Stuxnet. The captured memory file of this machine (*stuxnet.vmem*) is analyzed below.

First step in analysis is the identification of the profile, a necessary option required by Volatility in order to perform accurate results. This can be achieved by the *imageinfo* plugin:

```
python vol.py -f stuxnet.vmem imageinfo
```

which produces the following output:

```
Volatility Foundation Volatility Framework 2.5
INFO    : volatility.debug    : Determining profile based on KDBG search...
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with
WinXPSP2x86)
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (stuxnet.vmem)
                      PAE type : PAE
                           DTB : 0x319000L
                          KDBG : 0x80545ae0L
          Number of Processors : 1
     Image Type (Service Pack) : 3
                KPCR for CPU 0 : 0xffdff000L
             KUSER_SHARED_DATA : 0xffdf0000L
           Image date and time : 2011-06-03 04:31:36 UTC+0000
     Image local date and time : 2011-06-03 00:31:36 -0400
```

[41] https://virusshare.com
[42] http://openmalware.org/
[43] http://malshare.com/
[44] http://malwarecookbook.googlecode.com/svn/trunk/stuxnet.vmem.zip
[45] https://www.vmware.com/products/workstation

As a result, the profile used for the rest of the analysis is the one for Windows XP SP3 x86 operating system. One of the most common steps in a forensic analysis is the examination of the processes running on the system along with their handles, objects, modules, etc, while the memory capture was performed. This can be achieved by multiple plugins like *pslist*, *pstree*, *handles* and *psxview*.

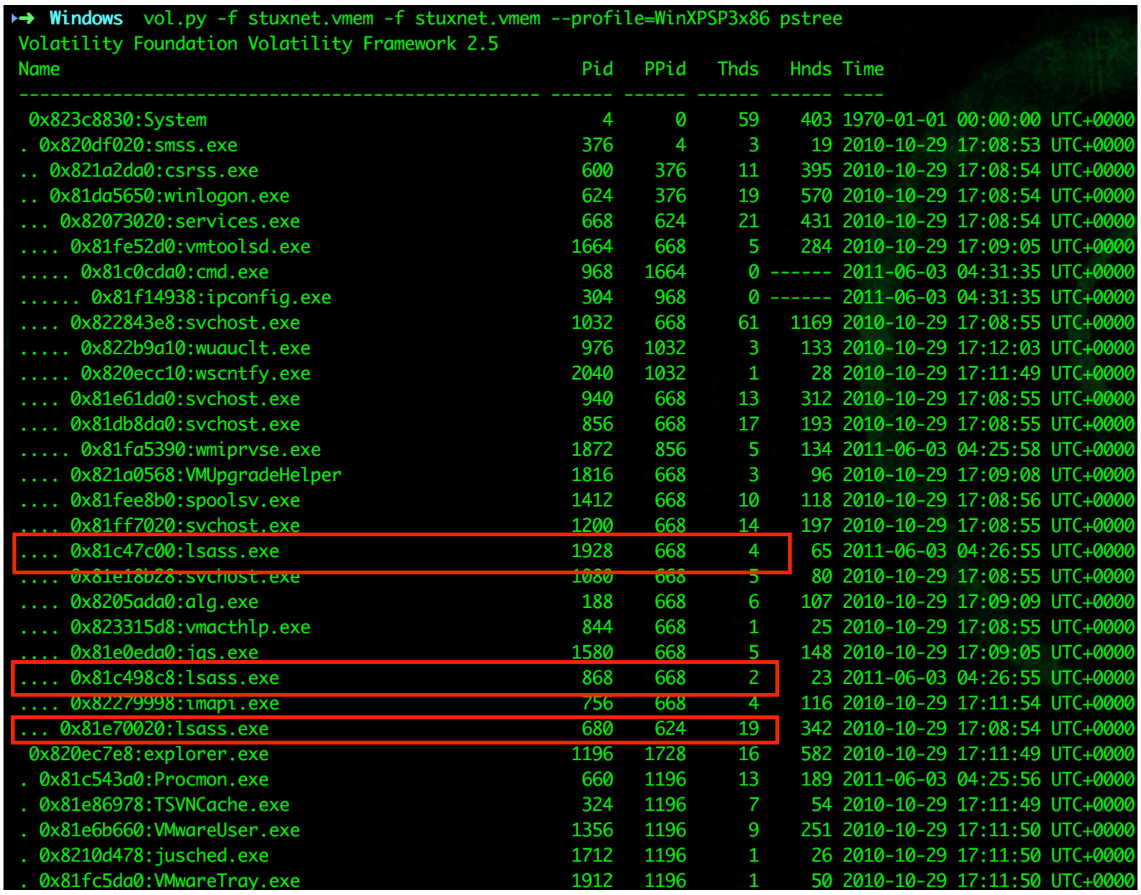
python vol.py –f stuxnet.vmem –profile= WinXPSP3x86 pstree

which produces the output shown in Figure 3-3. By examining the tree of processes there is an out of the ordinary number of processes posing as *lsass.exe*, a system service in Microsoft Windows responsible for the Local Security Authority Subsystem[46]. On an uninfected system there is only one instance of *lsass.exe*. Piping the result of Volatility to *egrep*[47] will produce an even clearer output:

python vol.py -f stuxnet.vmem -f stuxnet.vmem --profile=WinXPSP3x86 pstree | egrep '(lsass.exe)'

By focusing the investigation on those suspicious instances of *lsass.exe*, more clues of malicious behavior are revealed.

[46] https://technet.microsoft.com/en-us/library/cc961760.aspx
[47] http://linux.die.net/man/1/egrep



```
➜  Windows  vol.py -f stuxnet.vmem -f stuxnet.vmem --profile=WinXPSP3x86 pstree
Volatility Foundation Volatility Framework 2.5
Name                                                  Pid   PPid   Thds   Hnds Time
-------------------------------------------------- ------ ------ ------ ------ ----
 0x823c8830:System                                      4      0     59    403 1970-01-01 00:00:00 UTC+0000
. 0x820df020:smss.exe                                 376      4      3     19 2010-10-29 17:08:53 UTC+0000
.. 0x821a2da0:csrss.exe                               600    376     11    395 2010-10-29 17:08:54 UTC+0000
.. 0x81da5650:winlogon.exe                            624    376     19    570 2010-10-29 17:08:54 UTC+0000
... 0x82073020:services.exe                           668    624     21    431 2010-10-29 17:08:54 UTC+0000
.... 0x81fe52d0:vmtoolsd.exe                         1664    668      5    284 2010-10-29 17:09:05 UTC+0000
..... 0x81c0cda0:cmd.exe                              968   1664      0 ------ 2011-06-03 04:31:35 UTC+0000
...... 0x81f14938:ipconfig.exe                        304    968      0 ------ 2011-06-03 04:31:35 UTC+0000
.... 0x822843e8:svchost.exe                          1032    668     61   1169 2010-10-29 17:08:55 UTC+0000
..... 0x822b9a10:wuauclt.exe                          976   1032      3    133 2010-10-29 17:12:03 UTC+0000
..... 0x820ecc10:wscntfy.exe                         2040   1032      1     28 2010-10-29 17:11:49 UTC+0000
.... 0x81e61da0:svchost.exe                           940    668     13    312 2010-10-29 17:08:55 UTC+0000
.... 0x81db8da0:svchost.exe                           856    668     17    193 2010-10-29 17:08:55 UTC+0000
..... 0x81fa5390:wmiprvse.exe                        1872    856      5    134 2011-06-03 04:25:58 UTC+0000
.... 0x821a0568:VMUpgradeHelper                      1816    668      3     96 2010-10-29 17:09:08 UTC+0000
.... 0x81fee8b0:spoolsv.exe                          1412    668     10    118 2010-10-29 17:08:56 UTC+0000
.... 0x81ff7020:svchost.exe                          1200    668     14    197 2010-10-29 17:08:55 UTC+0000
.... 0x81c47c00:lsass.exe                            1928    668      4     65 2011-06-03 04:26:55 UTC+0000
.... 0x81e18b28:svchost.exe                          1080    668      5     80 2010-10-29 17:08:55 UTC+0000
.... 0x8205ada0:alg.exe                               188    668      6    107 2010-10-29 17:09:09 UTC+0000
.... 0x823315d8:vmacthlp.exe                          844    668      1     25 2010-10-29 17:08:55 UTC+0000
.... 0x81e0eda0:jqs.exe                              1580    668      5    148 2010-10-29 17:09:05 UTC+0000
.... 0x81c498c8:lsass.exe                             868    668      2     23 2011-06-03 04:26:55 UTC+0000
.... 0x82279998:imapi.exe                             756    668      4    116 2010-10-29 17:11:54 UTC+0000
... 0x81e70020:lsass.exe                              680    624     19    342 2010-10-29 17:08:54 UTC+0000
 0x820ec7e8:explorer.exe                             1196   1728     16    582 2010-10-29 17:11:49 UTC+0000
. 0x81c543a0:Procmon.exe                              660   1196     13    189 2011-06-03 04:25:56 UTC+0000
. 0x81e86978:TSVNCache.exe                            324   1196      7     54 2010-10-29 17:11:49 UTC+0000
. 0x81e6b660:VMwareUser.exe                          1356   1196      9    251 2010-10-29 17:11:50 UTC+0000
. 0x8210d478:jusched.exe                             1712   1196      1     26 2010-10-29 17:11:50 UTC+0000
. 0x81fc5da0:VMwareTray.exe                          1912   1196      1     50 2010-10-29 17:11:50 UTC+0000
```

Figure 3-3, Executing the *pstree* plugin on a Stuxnet infected system

One of the three instances has a start time in 2010 while the other two in 2011. Considering the fact that every other program has started in 2010, the latter instances of *lsass.exe* are definitely out of the order.

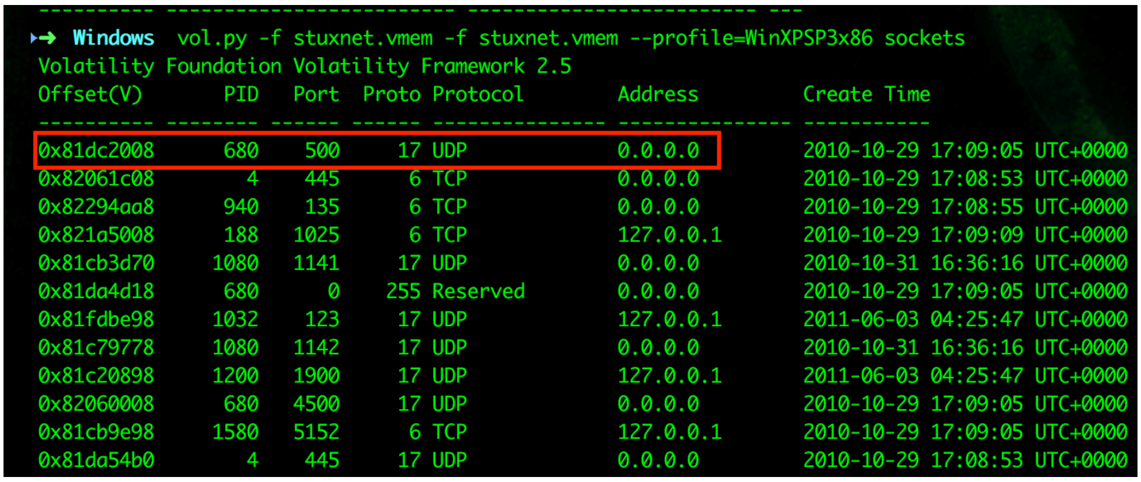
Another common step when analyzing malicious samples is the examination of network connections of the operating system. This may not be a clear indication of malicious behavior since at the time the memory capture occurred there may be no network connections present. However, it is always a measure of good practice to take this step under consideration and use the *connections* plugin:

python vol.py -f stuxnet.vmem -f stuxnet.vmem --profile=WinXPSP3x86 connections

In this case, there are no network connections at the time the memory was captured. There is another plugin however that will help detect malicious behavior regarding connections to a network, the *sockets* plugin, responsible for printing a list of open network sockets[48].

python vol.py -f stuxnet.vmem -f stuxnet.vmem --profile=WinXPSP3x86 sockets

The output is shown in Figure 3-4 below. Again, there is no sign of malicious activity.



```
---------- ------------------------ ------------------------ ---
➜ Windows  vol.py -f stuxnet.vmem -f stuxnet.vmem --profile=WinXPSP3x86 sockets
Volatility Foundation Volatility Framework 2.5
Offset(V)       PID   Port  Proto Protocol        Address         Create Time
---------- -------- ------ ------ --------------- --------------- -----------
0x81dc2008      680    500     17 UDP             0.0.0.0         2010-10-29 17:09:05 UTC+0000
0x82061c08        4    445      6 TCP             0.0.0.0         2010-10-29 17:08:53 UTC+0000
0x82294aa8      940    135      6 TCP             0.0.0.0         2010-10-29 17:08:55 UTC+0000
0x821a5008      188   1025      6 TCP             127.0.0.1       2010-10-29 17:09:09 UTC+0000
0x81cb3d70     1080   1141     17 UDP             0.0.0.0         2010-10-31 16:36:16 UTC+0000
0x81da4d18      680      0    255 Reserved        0.0.0.0         2010-10-29 17:09:05 UTC+0000
0x81fdbe98     1032    123     17 UDP             127.0.0.1       2011-06-03 04:25:47 UTC+0000
0x81c79778     1080   1142     17 UDP             0.0.0.0         2010-10-31 16:36:16 UTC+0000
0x81c20898     1200   1900     17 UDP             127.0.0.1       2011-06-03 04:25:47 UTC+0000
0x82060008      680   4500     17 UDP             0.0.0.0         2010-10-29 17:09:05 UTC+0000
0x81cb9e98     1580   5152      6 TCP             127.0.0.1       2010-10-29 17:09:05 UTC+0000
0x81da54b0        4    445     17 UDP             0.0.0.0         2010-10-29 17:08:53 UTC+0000
```

Figure 3-4, Output of the *connections* plugin

But in conjunction with the results of *pstree* this absence of some specific network elements is itself a sign of malicious activity; The highlighted socket in Figure 3-4 belongs to a process with a PID equal to one of the three instances of *lsass.exe* we noticed earlier. The process is listening to port 500 and 4500 which according to Microsoft is normal behavior. Thus, the two other instances of *lsass.exe* that do not experience such behavior are not acting as they should.

Another indicator of a malicious process, especially in Windows systems, is the number of DLLs[49] it is depending on for its functioning. Usually a typical program requires about 40 DLLs or more. By using the *dlllist* plugin in Volatility to investigate the two suspicious instances of *lsass.exe*, we can print a list of loaded DLLs for each process.

python vol.py -f stuxnet.vmem -f stuxnet.vmem --profile=WinXPSP3x86 -p 680 dlllist

where *–p 680* is the PID of the lsass.exe instance with timestamp in 2010. The result is 64 DLLs which is to be expected. Running the plugin against the two other instances, the

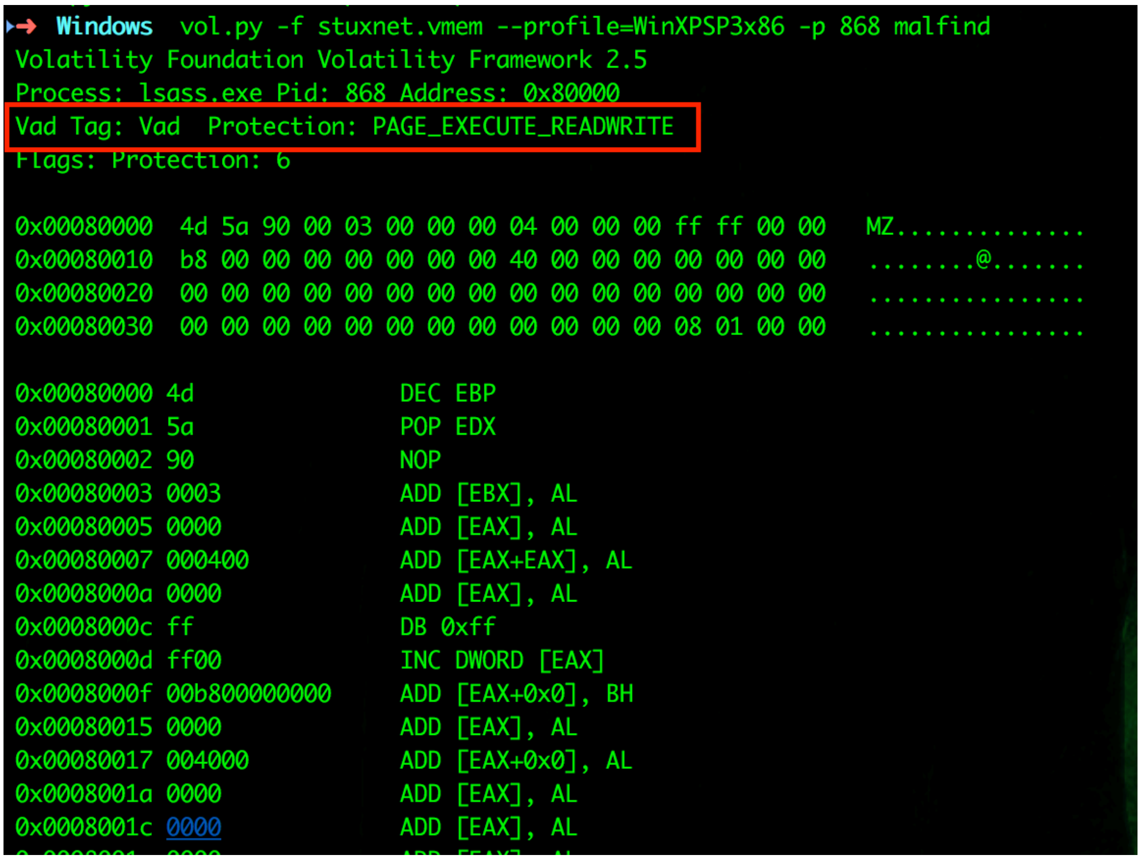
[48] https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html
[49] https://support.microsoft.com/en-us/kb/815065

number is significantly lower; 15 and 35. Which is a third, strong indication that these processes are not legitimate.

The next plugin we are going to execute against the sample is *malfind*[50], a plugin dedicated in locating and flagging patterns in memory that indicate malicious behavior. In order to target a specific process, we pass the corresponding PID(s) to *malfind:*

vol.py -f stuxnet.vmem --profile=WinXPSP3x86 -p 868 malfind

which produces this output:



```
➜ Windows  vol.py -f stuxnet.vmem --profile=WinXPSP3x86 -p 868 malfind
Volatility Foundation Volatility Framework 2.5
Process: lsass.exe Pid: 868 Address: 0x80000
Vad Tag: Vad  Protection: PAGE_EXECUTE_READWRITE
Flags: Protection: 6

0x00080000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00080010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00080020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00080030  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00   ................

0x00080000 4d               DEC EBP
0x00080001 5a               POP EDX
0x00080002 90               NOP
0x00080003 0003             ADD [EBX], AL
0x00080005 0000             ADD [EAX], AL
0x00080007 000400           ADD [EAX+EAX], AL
0x0008000a 0000             ADD [EAX], AL
0x0008000c ff               DB 0xff
0x0008000d ff00             INC DWORD [EAX]
0x0008000f 00b800000000     ADD [EAX+0x0], BH
0x00080015 0000             ADD [EAX], AL
0x00080017 004000           ADD [EAX+0x0], AL
0x0008001a 0000             ADD [EAX], AL
0x0008001c 0000             ADD [EAX], AL
```

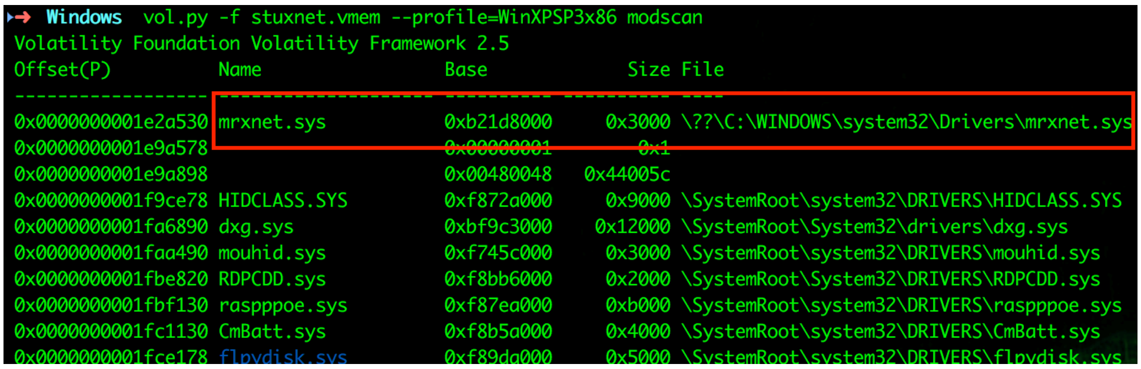
Figure 3-5, Executing *malfind* plugin against the malicious process

According to the plugin there are areas in memory that are marked with READ, WRITE and EXECUTE permissions, as highlighted above. A normal, non-malicious executable must have READ and EXECUTE permissions and should not be able to write in any part

[50] https://code.google.com/p/volatility/wiki/CommandReferenceMal22#malfind

of the memory. *Malfind* also provided the ability to export the malicious content for further analysis via a disassembler or antivirus services via the *--dump-dir {directory}* switch. In a similar pattern, there is the *procdump* plugin used for dumping a process to an executable file sample.

Volatility provides another two core plugins that can help us pinpoint any files related to malware, *modscan* and *moddump*. The first one can be used to scan the sample for drivers and other kernel modules, while the latter can be used to export those for further analysis.



```
➜ Windows  vol.py -f stuxnet.vmem --profile=WinXPSP3x86 modscan
Volatility Foundation Volatility Framework 2.5
Offset(P)          Name                 Base             Size File
------------------ -------------------- ---------- ---------- ----
0x000000001e2a530 mrxnet.sys           0xb21d8000     0x3000 \??\C:\WINDOWS\system32\Drivers\mrxnet.sys
0x000000001e9a578                      0x00000001        0x1
0x000000001e9a898                      0x00480048   0x44005c
0x000000001f9ce78 HIDCLASS.SYS         0xf872a000     0x9000 \SystemRoot\system32\DRIVERS\HIDCLASS.SYS
0x000000001fa6890 dxg.sys              0xbf9c3000    0x12000 \SystemRoot\System32\drivers\dxg.sys
0x000000001faa490 mouhid.sys           0xf745c000     0x3000 \SystemRoot\system32\DRIVERS\mouhid.sys
0x000000001fbe820 RDPCDD.sys           0xf8bb6000     0x2000 \SystemRoot\System32\DRIVERS\RDPCDD.sys
0x000000001fbf130 raspppoe.sys         0xf87ea000     0xb000 \SystemRoot\system32\DRIVERS\raspppoe.sys
0x000000001fc1130 CmBatt.sys           0xf8b5a000     0x4000 \SystemRoot\system32\DRIVERS\CmBatt.sys
0x000000001fce178 flpydisk.sys         0xf89da000     0x5000 \SystemRoot\system32\DRIVERS\flpydisk.sys
```

Figure 3-6, Volatility plugin *modscan* into action

The result is a long list of modules and drivers, a couple of whom are located in a non-common location, as shown in Figure 3-6. *Moddump* plugin can be used to export them for further analysis. In addition, since we now hold a reference to the name of a possibly malicious driver, we can examine any references to it in the Windows Registry[51], via the *strings*[52][53] program.

strings stuxnet.vmem | grep mrx -i | grep HKLM[54]

This will result to an output where everything related to Windows Registry will be printed and can be further manipulated again by piping the results to *grep*.

python vol.py -f stuxnet.vmem printkey -K 'ControlSet001\Services\MrxNet'

python vol.py -f stuxnet.vmem printkey -K 'ControlSet001\Services\MrxCls'

[51] https://support.microsoft.com/en-us/kb/256986
[52] http://linux.die.net/man/1/strings
[53] https://technet.microsoft.com/en-us/sysinternals/strings.aspx
[54] http://www.computerhope.com/jargon/h/hklm.htm

Finally, by using the *printkey* plugin which is used to print a registry key along with is values and subkeys, we can verify that Stuxnet starts executing every time the operating systems reboots by locating the appropriate registry key.

3.5.2 Linux

Linux may not be as popular target as Windows are when it comes to malware, but that should not be advocated as reason for Linux being a platform with complete lack of malware. On the contrary due to the fact that Linux is installed in the majority of servers on the Internet[57] and Google's Android, which is based on Linux, has became the most used operating system globally[58][44], instances of malware are on the rise and forensic analysis tools should be prepared.

Volatility has more that 65 plugins dedicated to memory captures that are derived from Linux. The list of those plugins can be printed by grepping the *info* output for *linux*:

```
python vol.py --info | grep linux
```

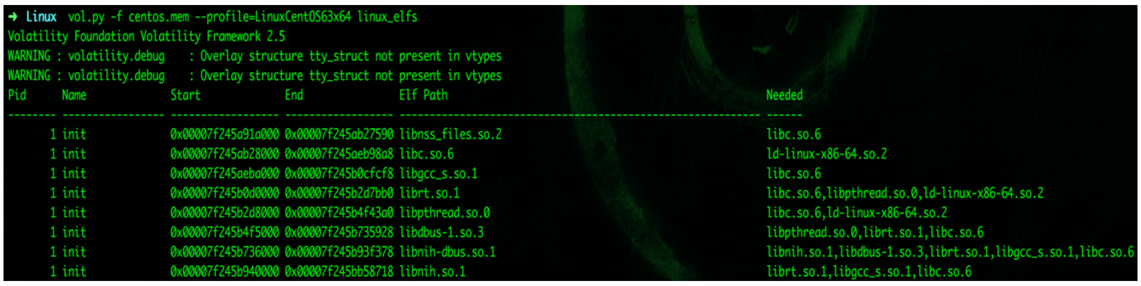
Following we are going to demonstrate the usage of some of those plugins to a memory capture taken from a system running Centos 6.3 x64[55].

Just like the previous case study when examining a Windows memory capture, we start by enumerating the processes of the system

```
python vol.py -f centos.mem --profile=LinuxCentOS63x64 linux_pstree
```

The output is a quite long list which can be further manipulated by piping the results to *grep* in order to produce a friendlier output. Checking information related to network connections and interfaces can be achieved by *linux_netscan*, *linux_netstat*, and *linux_ifconfig* respectively. Another useful plugin when suspecting a malicious executable is *linux_elfs* which locates ELF binaries in process mappings.

[55] https://secondlookforensics.com/linux-memory-images/

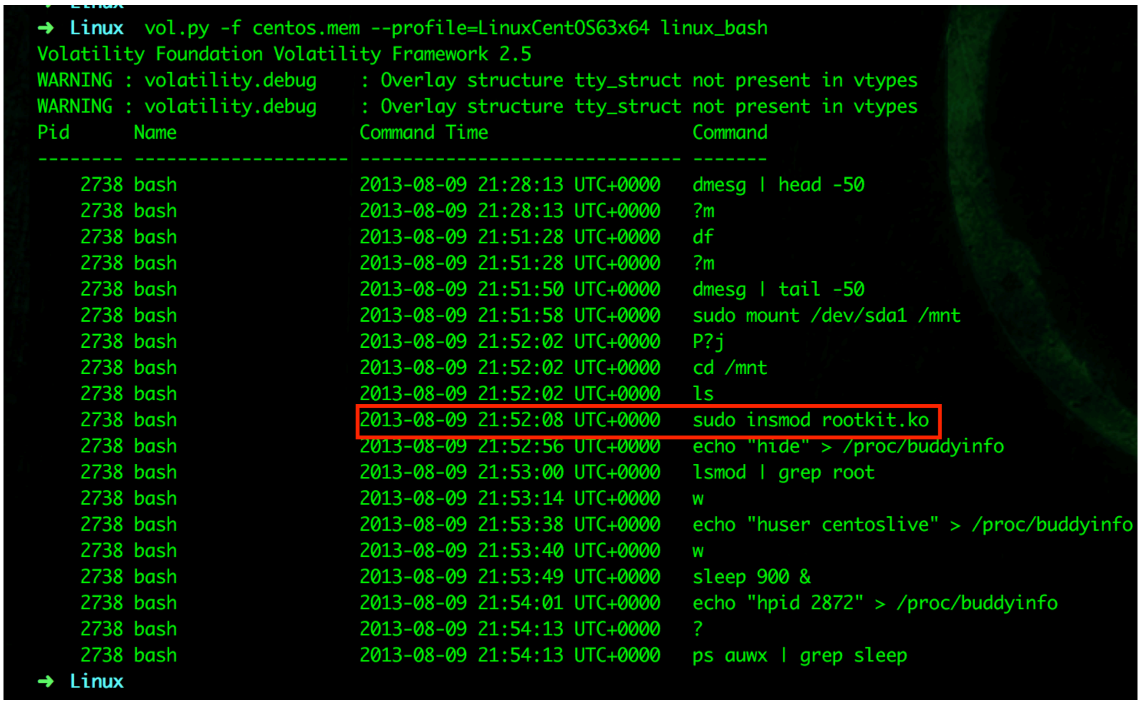


```
➜ Linux  vol.py -f centos.mem --profile=LinuxCentOS63x64 linux_elfs
Volatility Foundation Volatility Framework 2.5
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
Pid      Name             Start              End                Elf Path                                                       Needed
-------- ---------------- ------------------ ------------------ -------------------------------------------------------------- ------
       1 init             0x00007f245a91a000 0x00007f245ab27590 libnss_files.so.2                                              libc.so.6
       1 init             0x00007f245ab28000 0x00007f245aeb98a8 libc.so.6                                                      ld-linux-x86-64.so.2
       1 init             0x00007f245aeba000 0x00007f245b0cfcf8 libgcc_s.so.1                                                  libc.so.6
       1 init             0x00007f245b0d0000 0x00007f245b2d7bb0 librt.so.1                                                     libc.so.6,libpthread.so.0,ld-linux-x86-64.so.2
       1 init             0x00007f245b2d8000 0x00007f245b4f43a0 libpthread.so.0                                                libc.so.6,ld-linux-x86-64.so.2
       1 init             0x00007f245b4f5000 0x00007f245b735928 libdbus-1.so.3                                                 libpthread.so.0,librt.so.1,libc.so.6
       1 init             0x00007f245b736000 0x00007f245b93f378 libnih-dbus.so.1                                               libnih.so.1,libdbus-1.so.3,librt.so.1,libgcc_s.so.1,libc.so.6
       1 init             0x00007f245b940000 0x00007f245bb58718 libnih.so.1                                                    librt.so.1,libgcc_s.so.1,libc.so.6
```

Figure 3-7, Locating ELF binaries via the *linux_elfs* plugin

When performing malicious actions on a Linux system, an advanced use is well aware that every command she uses is stored in bash[56] profile history and must take precautions in order to avoid this feature. Such precautions may be to manipulate the history file (HISTFILE)[57] or when logging in via secure shell to use the T switch, which disables the pseudo-terminal allocation[58]. However even when some or all of these measures are taken, bash history can still be retrieved by Volatility since the commands have to exist at memory at some point.

[56] https://www.gnu.org/software/bash/
[57] https://www.gnu.org/software/bash/manual/html_node/Bash-History-Facilities.html
[58] http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man1/ssh.1?query=ssh

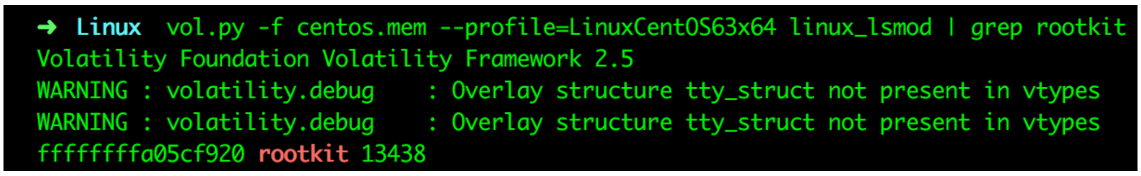


```
➜ Linux  vol.py -f centos.mem --profile=LinuxCentOS63x64 linux_bash
Volatility Foundation Volatility Framework 2.5
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
Pid      Name                 Command Time                   Command
-------- -------------------- ------------------------------ -------
    2738 bash                 2013-08-09 21:28:13 UTC+0000   dmesg | head -50
    2738 bash                 2013-08-09 21:28:13 UTC+0000   ?m
    2738 bash                 2013-08-09 21:51:28 UTC+0000   df
    2738 bash                 2013-08-09 21:51:28 UTC+0000   ?m
    2738 bash                 2013-08-09 21:51:50 UTC+0000   dmesg | tail -50
    2738 bash                 2013-08-09 21:51:58 UTC+0000   sudo mount /dev/sda1 /mnt
    2738 bash                 2013-08-09 21:52:02 UTC+0000   P?j
    2738 bash                 2013-08-09 21:52:02 UTC+0000   cd /mnt
    2738 bash                 2013-08-09 21:52:02 UTC+0000   ls
    2738 bash                 2013-08-09 21:52:08 UTC+0000   sudo insmod rootkit.ko
    2738 bash                 2013-08-09 21:52:56 UTC+0000   echo "hide" > /proc/buddyinfo
    2738 bash                 2013-08-09 21:53:00 UTC+0000   lsmod | grep root
    2738 bash                 2013-08-09 21:53:14 UTC+0000   w
    2738 bash                 2013-08-09 21:53:38 UTC+0000   echo "huser centoslive" > /proc/buddyinfo
    2738 bash                 2013-08-09 21:53:40 UTC+0000   w
    2738 bash                 2013-08-09 21:53:49 UTC+0000   sleep 900 &
    2738 bash                 2013-08-09 21:54:01 UTC+0000   echo "hpid 2872" > /proc/buddyinfo
    2738 bash                 2013-08-09 21:54:13 UTC+0000   ?
    2738 bash                 2013-08-09 21:54:13 UTC+0000   ps auwx | grep sleep
➜ Linux
```

Figure 3-8, Retrieving bash history via the *linux_bash* plugin

In this case the creator of a Linux rootkit[59] inserted it into the kernel and later hide it from *lsmod*[59], a program used to show the status of modules in the Linux Kernel. Although the creator of the rootkit was clever enough to hide its presence from user space [60], tools like Volatility which can analyse user space and kernel space, are able to detect it:

python vol.py -f centos.mem --profile=LinuxCentOS63x64 linux_lsmod | grep rootkit



```
➜ Linux  vol.py -f centos.mem --profile=LinuxCentOS63x64 linux_lsmod | grep rootkit
Volatility Foundation Volatility Framework 2.5
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
WARNING : volatility.debug    : Overlay structure tty_struct not present in vtypes
ffffffffa05cf920 rootkit 13438
```

Figure 3-9, Using *linux_lsmod* to detect loaded kernel modules

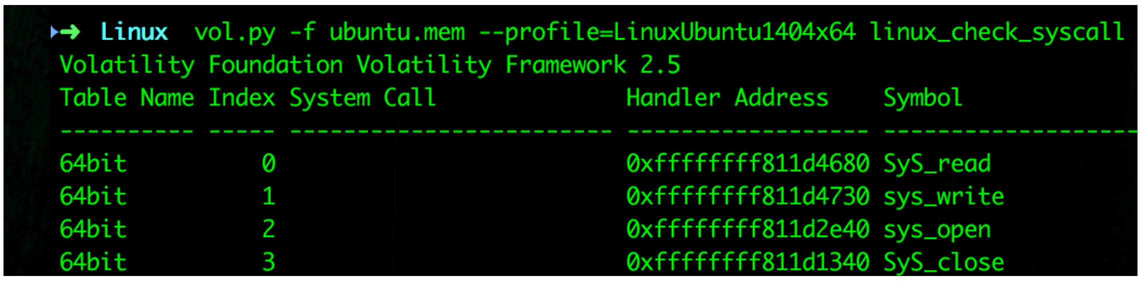
There are cases though that the creator of the malware may hide its presence from *lsmod* even when kernel space is checked. For this kind of manipulations there is the *sysfs* file system [61], a virtual file system that exports information about various parts of the sys-

[59] http://linux.die.net/man/8/lsmod

tem including hardware, kernel subsystems and everything related to device drivers. Volatility has a Linux plugin dedicated to using *sysfs* for gathering all the modules loaded into the kernel[21][62]:

python vol.py -f ubuntu.mem --profile=LinuxUbuntu1404x64 linux_check_module

One more way for a malicious user to hide her activity is to hook systems calls like *read*, *write, kill,* etc. which can be traced via the *linux_check_call* plugin.

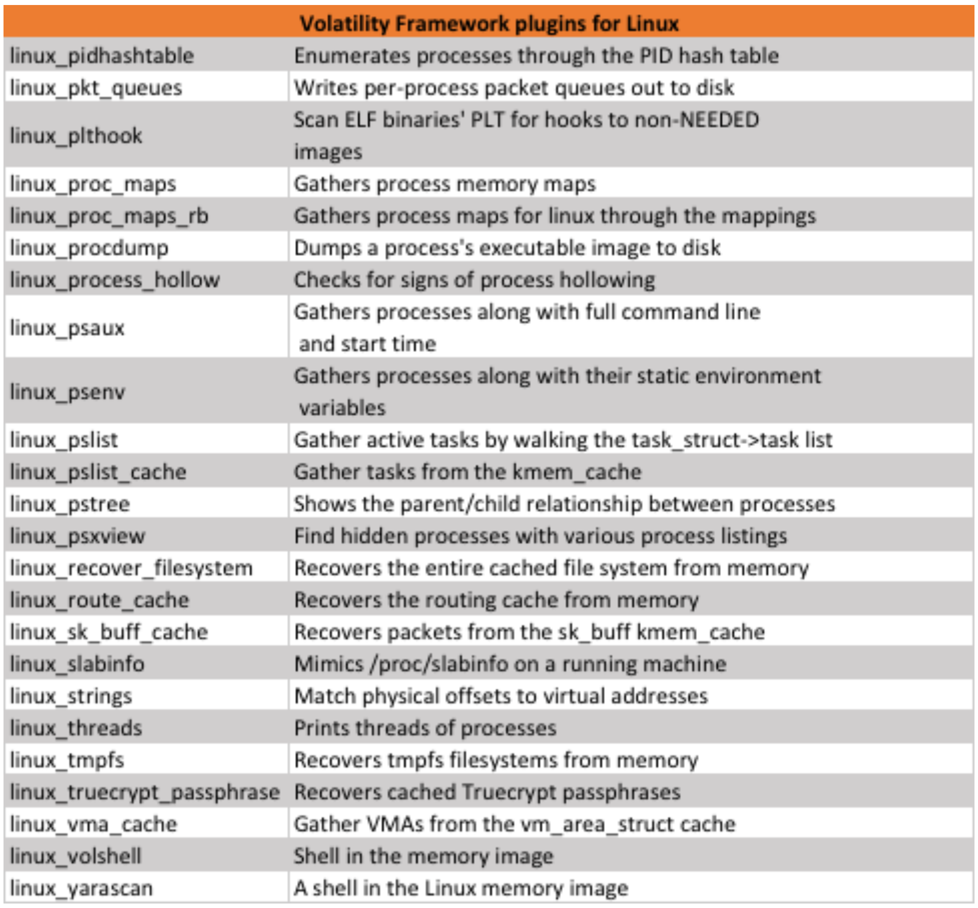


```
→ Linux  vol.py -f ubuntu.mem --profile=LinuxUbuntu1404x64 linux_check_syscall
Volatility Foundation Volatility Framework 2.5
Table Name Index System Call              Handler Address    Symbol
---------- ----- ------------------------ ------------------ ------------------
64bit          0                          0xffffffff811d4680 SyS_read
64bit          1                          0xffffffff811d4730 sys_write
64bit          2                          0xffffffff811d2e40 sys_open
64bit          3                          0xffffffff811d1340 SyS_close
```

Figure 3-10, Looking into system calls via the *linux_check_call* plugin

Volatility Framework plugins for Linux	
linux_apihooks	Checks for userland apihooks
linux_arp	Print the ARP table
linux_banner	Prints the Linux banner information
linux_bash	Recover bash history from bash process memory
linux_bash_env	Recover a process' dynamic environment variables
linux_bash_hash	Recover bash hash table from bash process memory
linux_check_afinfo	Verifies the operation function pointers of network protocols
linux_check_creds	Checks if any processes are sharing credential structures
linux_check_evt_arm	Checks the Exception Vector Table to look for syscall table hooking
linux_check_fop	Check file operation structures for rootkit modifications
linux_check_idt	Checks if the IDT has been altered
linux_check_inline_kernel	Check for inline kernel hooks
linux_check_modules	Compares module list to sysfs info, if available
linux_check_syscall	Checks if the system call table has been altered
linux_check_syscall_arm	Checks if the system call table has been altered
linux_check_tty	Checks tty devices for hooks
linux_cpuinfo	Prints info about each active processor
linux_dentry_cache	Gather files from the dentry cache
linux_dmesg	Gather dmesg buffer
linux_dump_map	Writes selected memory mappings to disk
linux_dynamic_env	Recover a process' dynamic environment variables
linux_elfs	Find ELF binaries in process mappings
linux_enumerate_files	Lists files referenced by the filesystem cache
linux_find_file	Lists and recovers files from memory
linux_getcwd	Lists current working directory of each process
linux_hidden_modules	Carves memory to find hidden kernel modules
linux_ifconfig	Gathers active interfaces
linux_info_regs	It's like 'info registers' in GDB
linux_iomem	Provides output similar to /proc/iomem
linux_kernel_opened_files	Lists files that are opened from within the kernel
linux_keyboard_notifiers	Parses the keyboard notifier call chain
linux_ldrmodules	Compares the output of proc maps with the list of libraries from libdl
linux_library_list	Lists libraries loaded into a process
linux_librarydump	Dumps shared libraries in process memory to disk
linux_list_raw	List applications with promiscuous sockets
linux_lsmod	Gather loaded kernel modules
linux_lsof	Lists file descriptors and their path
linux_malfind	Looks for suspicious process mappings
linux_memmap	Dumps the memory map for linux tasks
linux_moddump	Extract loaded kernel modules
linux_mount	Gather mounted fs/devices
linux_mount_cache	Gather mounted fs/devices from kmem_cache
linux_netfilter	Lists Netfilter hooks
linux_netscan	Carves for network connection structures
linux_netstat	Lists open sockets

Table 3-2, Volatility Framework plugins dedicated to Linux



Volatility Framework plugins for Linux	
linux_pidhashtable	Enumerates processes through the PID hash table
linux_pkt_queues	Writes per-process packet queues out to disk
linux_plthook	Scan ELF binaries' PLT for hooks to non-NEEDED images
linux_proc_maps	Gathers process memory maps
linux_proc_maps_rb	Gathers process maps for linux through the mappings
linux_procdump	Dumps a process's executable image to disk
linux_process_hollow	Checks for signs of process hollowing
linux_psaux	Gathers processes along with full command line and start time
linux_psenv	Gathers processes along with their static environment variables
linux_pslist	Gather active tasks by walking the task_struct->task list
linux_pslist_cache	Gather tasks from the kmem_cache
linux_pstree	Shows the parent/child relationship between processes
linux_psxview	Find hidden processes with various process listings
linux_recover_filesystem	Recovers the entire cached file system from memory
linux_route_cache	Recovers the routing cache from memory
linux_sk_buff_cache	Recovers packets from the sk_buff kmem_cache
linux_slabinfo	Mimics /proc/slabinfo on a running machine
linux_strings	Match physical offsets to virtual addresses
linux_threads	Prints threads of processes
linux_tmpfs	Recovers tmpfs filesystems from memory
linux_truecrypt_passphrase	Recovers cached Truecrypt passphrases
linux_vma_cache	Gather VMAs from the vm_area_struct cache
linux_volshell	Shell in the memory image
linux_yarascan	A shell in the Linux memory image

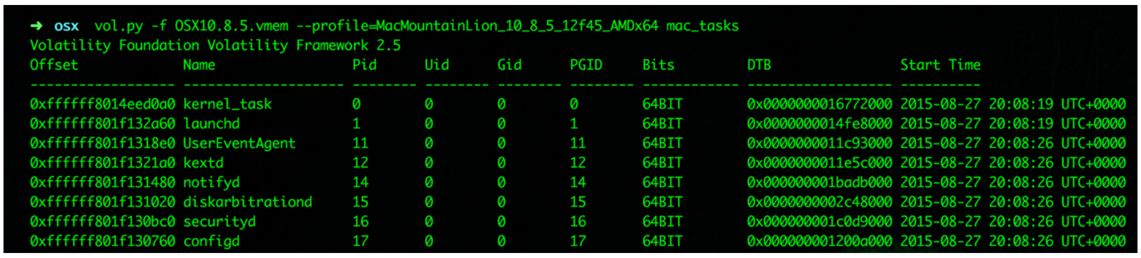
Table 3-3, Volatility Framework plugins dedicated to Linux

As in the Windows case study, usage of a combination of plugins along with critical thinking can help a forensic analyst detect abnormal behaviour and dig further to pinpoint the root of the problem.

3.5.3 OS X

As in the previous case studies, most of the advanced malware targeting users of Apple's operating systems operates only in memory to avoid detection and make its analysis harder, making Volatility Framework a crucial tool for the forensic analyst.

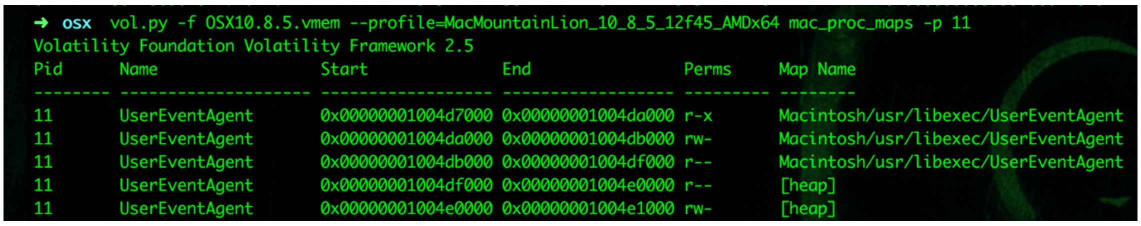
There are a handful of plugins for process enumeration, like *mac_pslist, mac_tasks* and *mac_pstree*, which can be really helpful for detecting methods of malware persistence on OS X described by P.Wardle [63][64].



```
➜ osx  vol.py -f OSX10.8.5.vmem --profile=MacMountainLion_10_8_5_12f45_AMDx64 mac_tasks
Volatility Foundation Volatility Framework 2.5
Offset             Name                 Pid      Uid      Gid      PGID     Bits         DTB                Start Time
------------------ -------------------- -------- -------- -------- -------- ------------ ------------------ ----------
0xffffff8014eed0a0 kernel_task          0        0        0        0        64BIT        0x0000000016772000 2015-08-27 20:08:19 UTC+0000
0xffffff801f132a60 launchd              1        0        0        1        64BIT        0x0000000014fe8000 2015-08-27 20:08:19 UTC+0000
0xffffff801f1318e0 UserEventAgent       11       0        0        11       64BIT        0x0000000011c93000 2015-08-27 20:08:26 UTC+0000
0xffffff801f1321a0 kextd                12       0        0        12       64BIT        0x0000000011e5c000 2015-08-27 20:08:26 UTC+0000
0xffffff801f131480 notifyd              14       0        0        14       64BIT        0x000000001badb000 2015-08-27 20:08:26 UTC+0000
0xffffff801f131020 diskarbitrationd     15       0        0        15       64BIT        0x0000000002c48000 2015-08-27 20:08:26 UTC+0000
0xffffff801f130bc0 securityd            16       0        0        16       64BIT        0x000000001c0d9000 2015-08-27 20:08:26 UTC+0000
0xffffff801f130760 configd              17       0        0        17       64BIT        0x000000001200a000 2015-08-27 20:08:26 UTC+0000
```

Figure 3-11, *mac_tasks* plugin provides a complete list of processes running on OSX along with attributes like PID, architecture and start time

When a particular process is suspicious, an investigator can retrieve details like the memory mapping of said process by using the *mac_proc_maps* plugin and providing the PID



```
➜ osx  vol.py -f OSX10.8.5.vmem --profile=MacMountainLion_10_8_5_12f45_AMDx64 mac_proc_maps -p 11
Volatility Foundation Volatility Framework 2.5
Pid      Name                 Start              End                Perms     Map Name
-------- -------------------- ------------------ ------------------ --------- --------
11       UserEventAgent       0x00000001004d7000 0x00000001004da000 r-x       Macintosh/usr/libexec/UserEventAgent
11       UserEventAgent       0x00000001004da000 0x00000001004db000 rw-       Macintosh/usr/libexec/UserEventAgent
11       UserEventAgent       0x00000001004db000 0x00000001004df000 r--       Macintosh/usr/libexec/UserEventAgent
11       UserEventAgent       0x00000001004df000 0x00000001004e0000 r--       [heap]
11       UserEventAgent       0x00000001004e0000 0x00000001004e1000 rw-       [heap]
```

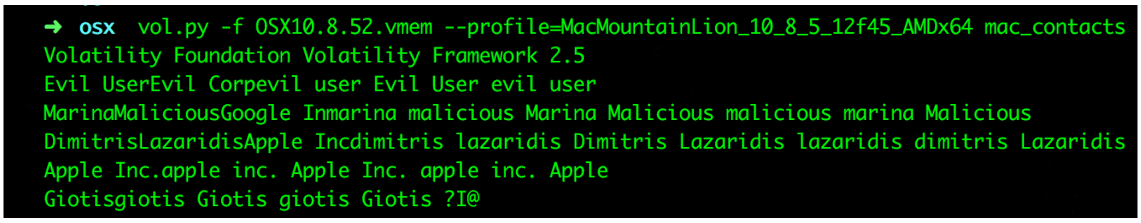
Figure 3-12, Mapping of *UserEventAgent* process

Moreover, *mac_handles* and *mac_threads* can provide an even deeper insight into the internals of a process.

As in other operating systems, network activity is always a factor of high importance when trying to detect malicious behavior. *Mac_netstat, mac_arp* and *mac_ifconfig* are the equivalent plugins of *linux_netstat, linux_arp* and *linux_ifconfig*, printing the network interfaces and network related activity.

There are also plugins like *mac_adium* and *mac_contacts* designed to extract information about the contacts of the user and his account registered to Adium, a popular instant messaging program for OS X[60]:

[60] https://adium.im/

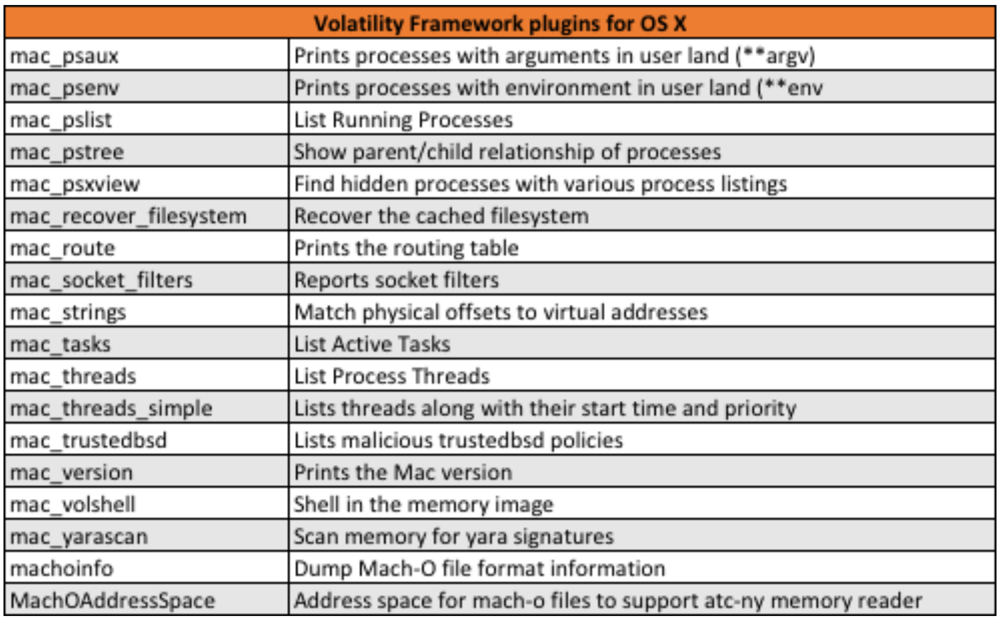


```
➜ osx  vol.py -f OSX10.8.52.vmem --profile=MacMountainLion_10_8_5_12f45_AMDx64 mac_contacts
Volatility Foundation Volatility Framework 2.5
Evil UserEvil Corpevil user Evil User evil user
MarinaMaliciousGoogle Inmarina malicious Marina Malicious malicious marina Malicious
DimitrisLazaridisApple Incdimitris lazaridis Dimitris Lazaridis lazaridis dimitris Lazaridis
Apple Inc.apple inc. Apple Inc. apple inc. Apple
Giotisgiotis Giotis giotis Giotis ?I@
```

Figure 3-13, Extracting OSX’s contact list with *mac_contacts* plugin

Volatility Framework plugins for OS X	
mac_adium	Lists Adium messages
mac_apihooks	Checks for API hooks in processes
mac_apihooks_kernel	Checks to see if system call and kernel functions are hooked
mac_arp	Prints the arp table
mac_bash	Recover bash history from bash process memory
mac_bash_env	Recover bash's environment variables
mac_bash_hash	Recover bash hash table from bash process memory
mac_calendar	Gets calendar events from Calendar.app
mac_check_mig_table	Lists entires in the kernel's MIG table
mac_check_syscall_shadow	Looks for shadow system call tables
mac_check_syscalls	Checks to see if system call table entries are hooked
mac_check_sysctl	Checks for unknown sysctl handlers
mac_check_trap_table	Checks to see if mach trap table entries are hooked
mac_compressed_swap	Prints Mac OS X VM compressor stats and dumps all compressed pages
mac_contacts	Gets contact names from Contacts.app
mac_dead_procs	Prints terminated/de-allocated processes
mac_dead_sockets	Prints terminated/de-allocated network sockets
mac_dead_vnodes	Lists freed vnode structures
mac_dmesg	Prints the kernel debug buffer
mac_dump_file	Dumps a specified file
mac_dump_maps	Dumps memory ranges of process(es)
mac_dyld_maps	Gets memory maps of processes from dyld data structur
mac_find_aslr_shift	Find the ASLR shift value for 10.8+ images
mac_get_profile	Automatically detect Mac profiles
mac_ifconfig	Lists network interface information for all devices
mac_ip_filters	Reports any hooked IP filters
mac_keychaindump	Recovers possbile keychain keys.
mac_ldrmodules	Compares the output of proc maps with the list of libraries from libdl
mac_librarydump	Dumps the executable of a process
mac_list_files	Lists files in the file cache
mac_list_kauth_listeners	Lists Kauth Scope listeners
mac_list_kauth_scopes	Lists Kauth Scopes and their status
mac_list_raw	List applications with promiscuous sockets
mac_list_sessions	Enumerates sessions
mac_list_zones	Prints active zones
mac_lsmod	Lists loaded kernel modules
mac_lsmod_iokit	Lists loaded kernel modules through IOkit
mac_lsmod_kext_map	Lists loaded kernel modules
mac_lsof	Lists per-process opened files
mac_machine_info	Prints machine information about the sample
mac_malfind	Looks for suspicious process mappings
mac_memdump	Dump addressable memory pages to a file
mac_moddump	Writes the specified kernel extension to disk
mac_mount	Prints mounted device information
mac_netstat	Lists active per-process network connections
mac_network_conns	Lists network connections from kernel network structures
mac_notesapp	Finds contents of Notes messages
mac_notifiers	Detects rootkits that add hooks into I/O Kit
mac_orphan_threads	Lists threads that don't map back to known modules/processes
mac_pgrp_hash_table	Walks the process group hash table
mac_pid_hash_table	Walks the pid hash table
mac_print_boot_cmdline	Prints kernel boot arguments
mac_proc_maps	Gets memory maps of processes
mac_procdump	Dumps the executable of a process

Table 3-4, Volatility Framework plugins dedicated to OS X



Volatility Framework plugins for OS X	
mac_psaux	Prints processes with arguments in user land (**argv)
mac_psenv	Prints processes with environment in user land (**env
mac_pslist	List Running Processes
mac_pstree	Show parent/child relationship of processes
mac_psxview	Find hidden processes with various process listings
mac_recover_filesystem	Recover the cached filesystem
mac_route	Prints the routing table
mac_socket_filters	Reports socket filters
mac_strings	Match physical offsets to virtual addresses
mac_tasks	List Active Tasks
mac_threads	List Process Threads
mac_threads_simple	Lists threads along with their start time and priority
mac_trustedbsd	Lists malicious trustedbsd policies
mac_version	Prints the Mac version
mac_volshell	Shell in the memory image
mac_yarascan	Scan memory for yara signatures
machoinfo	Dump Mach-O file format information
MachOAddressSpace	Address space for mach-o files to support atc-ny memory reader

Table 3-5, Volatility Framework plugins dedicated to OS X

In a recent paper presented in DFRWS 2015 by A.Case and G.Richard III [65], they introduced a series of new Volatility plugins, designed for detecting advanced rootkits for Mac OS X, a clear indication of the rapid expansion of The Volatility Framework but on the other hand, an strong clue of the unwanted attention the platform is attracting from malware authors.

3.6 Bitcoin Malware Case Study

Since its invention in 2008, Bitcoin has become the pinnacle of digital currencies [66]. One of the outcomes of this recognition was the increased attention of malicious users targeting legitimate users and their digital wallets. One of the ways a digital currency like Bitcoin differentiates from a physical currency, lies on the possible means of deception a legitimate user can be a victim of. Apart from stealing the actual Bitcoins, a malicious user can use the resources of a legitimate user (*mining*) to produce Bitcoins without the knowledge of the latter. This trend was on the rise until late 2013[6][7][4] but since Bitcoin mining has become uneconomic on home computers even if there is a collaboration of thousands of them[5], nowadays there is a decline of those kind of activities with the occasional news story every couple of months[67][35].

One of those illegal Bitcoin mining efforts in 2013 was spreading in Thessaloniki, Greece via USB sticks and later it passed the limits of the city and the country[61][62][63]. The malicious file responsible is a portable executable targeting Microsoft Windows, which is renaming itself to random strings, based on the contents of USB drives it detects. Upon execution it is replicating to *C:\Users\{Username}\AppData\Roaming* as *abab32.exe* along with a copy of a file named *sys32.exe*, which is a way of trying to avoid drawing attention if examined via the task manager. More over, to achieve persistence, it creates a registry entry to *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run* so as to execute every time the operating system reboots. While the first file turns out to be a legitimate Bitcoin miner[64], the original executable that extracted *abab32.exe* and *sys32.exe* into the system, falls into the definition of malware; It sneaks two executables into the system, tries achieves persistence of execution by modifying the registry and camouflages its icon to a the one of a folder in order misguide the unsuspicious user into executing it.

By reverse engineering the sample, a relationship diagram of its functions was produced:

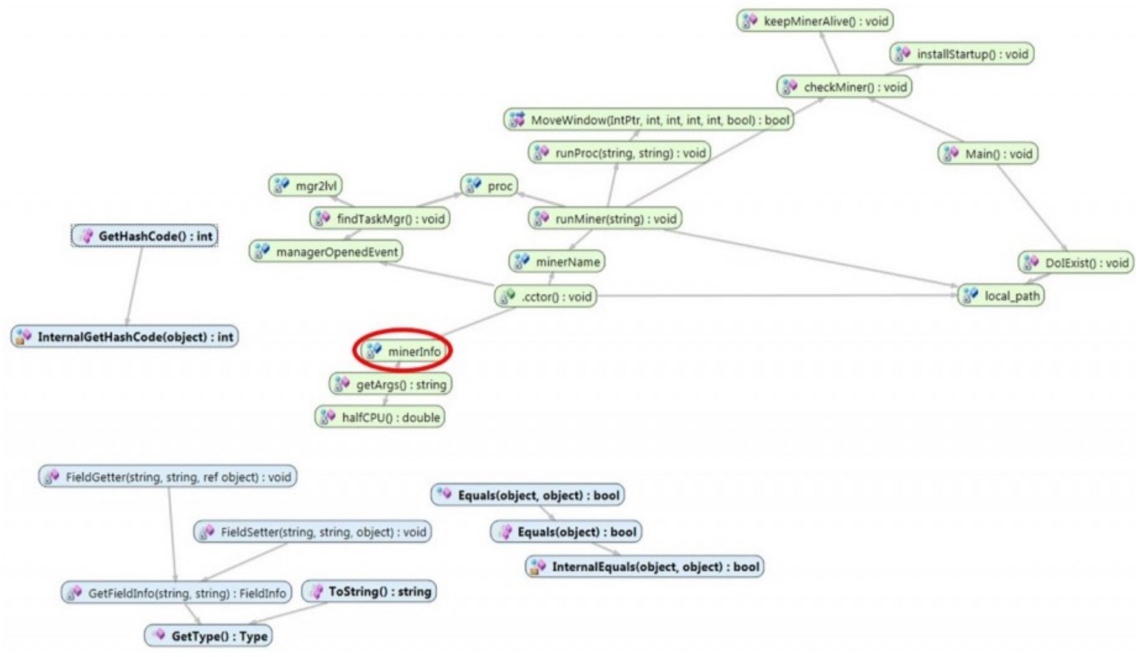
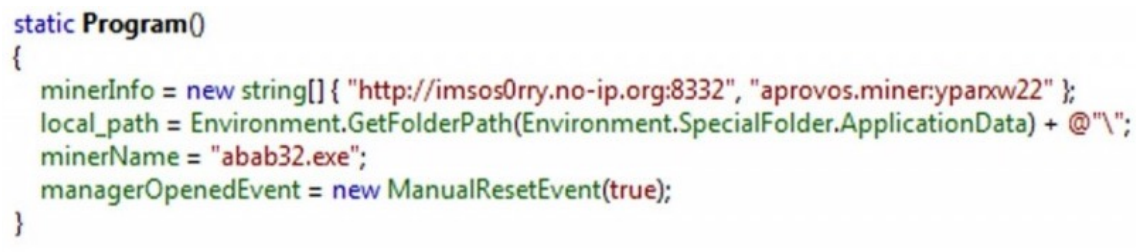


Figure 3-14, Functions of the malware after reverse engineering

[61] https://forums.malwarebytes.org/index.php?/topic/127472-remove-abab32exe-folders-turned-into-apps/
[62] http://www.thelab.gr/topic/122166-abab32exe/
[63] https://social.technet.microsoft.com/Forums/systemcenter/en-US/457b9bd3-65ba-4fbf-8fb3-35ee9357874f/appdataroamingabab32exe?forum=w7itprogeneral
[64] https://github.com/jgarzik/cpuminer

One of the most interesting functions of the malware highlighted in Figure 3-14 is *minerInfo*. After further analysis the source code of the function was available:

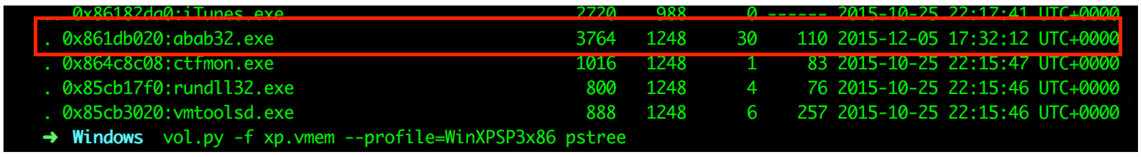


```
static Program()
{
    minerInfo = new string[] { "http://imsos0rry.no-ip.org:8332", "aprovos.miner:yparxw22" };
    local_path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + @"\";
    minerName = "abab32.exe";
    managerOpenedEvent = new ManualResetEvent(true);
}
```

Figure 3-15, Source code of *minerInfo* function

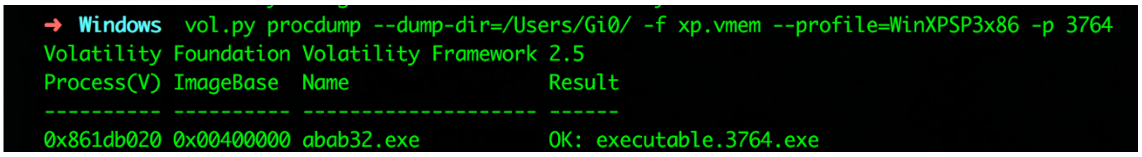
The malware after executing it connects to a specific domain and port, using a combination of username and password, awaiting for either pushing the results of its mining or receiving data for further mining.

Detection of this Bitcoin miner through Volatility relies on steps mentioned earlier in either of the case studies presented. Looking at the process list of the system via *pslist, pstree* or *psscan* and exporting suspicious processes to a file via *procdump* for further analysis like reverse engineering is one of the steps.



```
 .. 0x86182da0:iTunes.exe                        2720    988      0 ------ 2015-10-25 22:17:41 UTC+0000
 . 0x861db020:abab32.exe                         3764   1248     30    110 2015-12-05 17:32:12 UTC+0000
 . 0x864c8c08:ctfmon.exe                         1016   1248      1     83 2015-10-25 22:15:47 UTC+0000
 . 0x85cb17f0:rundll32.exe                        800   1248      4     76 2015-10-25 22:15:46 UTC+0000
 . 0x85cb3020:vmtoolsd.exe                        888   1248      6    257 2015-10-25 22:15:46 UTC+0000
➜ Windows  vol.py -f xp.vmem --profile=WinXPSP3x86 pstree
```

Figure 3-16 *Pstree* plugin locating the malicious process



```
➜ Windows  vol.py procdump --dump-dir=/Users/Gi0/ -f xp.vmem --profile=WinXPSP3x86 -p 3764
Volatility Foundation Volatility Framework 2.5
Process(V) ImageBase  Name                 Result
---------- ---------- -------------------- ------
0x861db020 0x00400000 abab32.exe           OK: executable.3764.exe
```

Figure 3-17, Exporting to a file via *procdump*

Moreover, examining everything related to network activity considering the nature of the malware and its need to connect to a server in order to operate, reveals a lot of information regarding its nature. By using *netscan* plugin a series of IPs are printed, information for whom can be obtained from sites like DNSstuff [65]or IPduh[66].

[65] http://www.dnsstuff.com/
[66] http://ipduh.com/

The username used by the miner in order to login to the remote server and the fact that the connection made were to IPs in Greece led to the identity of its creator, which resulted to an apology and seize of its operations[68][69].

3.7 Installation and setting up a test environment

The Volatility Framework supports all three major operating systems, Microsoft Windows, Apple OS X and Linux. In all cases there are two available means of installation; Building the framework from source code which is publicly available to everyone via git[67], a version control system created by Linus Torvalds[68], the creator of Linux, or using the binaries provided for each platform by the Volatility Foundation site[69]. If the user chooses the latter, the only prerequisite is Python 2.7[70], which for the Windows version is bundled into the installation executable. In the case of Linux and OS X, Pythons is already included into the operating system, thus simply running the binary is sufficient.

Using the source code for installation might add some overhead when upgrading or uninstalling if one used the *setup.py* file, but on the other hand it provides more inside into the inner workings of the framework, giving the opportunity to a researcher for further investigation and development.

In our case the test environment is based on OS X and Volatility Framework was installed via the source code, after downloading it via git and using the main Python file, *vol.py*:

git clone https://github.com/volatilityfoundation/volatility.git Volatility

After cloning the source code from git, the Volatility directory shown in Figure 3-18, contains everything needed for executing the framework. Particularly for OS X, apart from the source code and the binaries provided by the Foundation, there is an additional way of installation; a third party, free and open-source package manager for OS X, Homebrew[71]. Installation of Homebrew can be achieved via the terminal, with a single line:

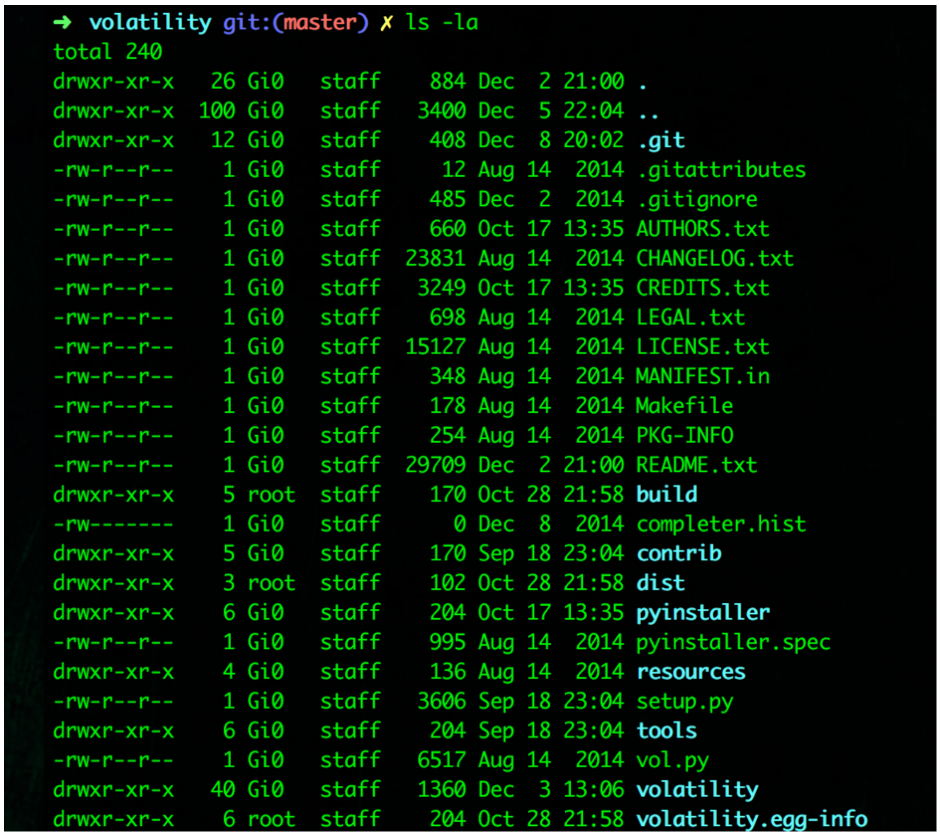
[67] https://github.com/volatilityfoundation/volatility
[68] https://www.britannica.com/biography/Linus-Torvalds
[69] http://www.volatilityfoundation.org/#!releases/component_71401
[70] https://www.python.org/download/releases/2.7/
[71] http://brew.sh/

ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

After this, Volatility can be installed by invoking Homebrew:

brew install volatility

In other words, Homebrew is the equivalent of the Apt package manager[72] in Debian Linux and its derivatives.



```
➜ volatility git:(master) ✗ ls -la
total 240
drwxr-xr-x   26 Gi0   staff    884 Dec  2 21:00 .
drwxr-xr-x  100 Gi0   staff   3400 Dec  5 22:04 ..
drwxr-xr-x   12 Gi0   staff    408 Dec  8 20:02 .git
-rw-r--r--    1 Gi0   staff     12 Aug 14  2014 .gitattributes
-rw-r--r--    1 Gi0   staff    485 Dec  2  2014 .gitignore
-rw-r--r--    1 Gi0   staff    660 Oct 17 13:35 AUTHORS.txt
-rw-r--r--    1 Gi0   staff  23831 Aug 14  2014 CHANGELOG.txt
-rw-r--r--    1 Gi0   staff   3249 Oct 17 13:35 CREDITS.txt
-rw-r--r--    1 Gi0   staff    698 Aug 14  2014 LEGAL.txt
-rw-r--r--    1 Gi0   staff  15127 Aug 14  2014 LICENSE.txt
-rw-r--r--    1 Gi0   staff    348 Aug 14  2014 MANIFEST.in
-rw-r--r--    1 Gi0   staff    178 Aug 14  2014 Makefile
-rw-r--r--    1 Gi0   staff    254 Aug 14  2014 PKG-INFO
-rw-r--r--    1 Gi0   staff  29709 Dec  2 21:00 README.txt
drwxr-xr-x    5 root  staff    170 Oct 28 21:58 build
-rw-------    1 Gi0   staff      0 Dec  8  2014 completer.hist
drwxr-xr-x    5 Gi0   staff    170 Sep 18 23:04 contrib
drwxr-xr-x    3 root  staff    102 Oct 28 21:58 dist
drwxr-xr-x    6 Gi0   staff    204 Oct 17 13:35 pyinstaller
-rw-r--r--    1 Gi0   staff    995 Aug 14  2014 pyinstaller.spec
drwxr-xr-x    4 Gi0   staff    136 Aug 14  2014 resources
-rw-r--r--    1 Gi0   staff   3606 Sep 18 23:04 setup.py
drwxr-xr-x    6 Gi0   staff    204 Sep 18 23:04 tools
-rw-r--r--    1 Gi0   staff   6517 Aug 14  2014 vol.py
drwxr-xr-x   40 Gi0   staff   1360 Dec  3 13:06 volatility
drwxr-xr-x    6 root  staff    204 Oct 28 21:58 volatility.egg-info
```

Figure 3-18, The Volatility Framework installation folder

Cloning the source code from *git* to a directory of our choice and running the main Python file *vol.py* from that directory has the disadvantage of not being able to use Volatility as a library but on the other hand it is easier to execute, upgrade or test multiple versions of it.

[72] https://wiki.debian.org/Apt

Although Volatility can be used to analyse files without the installation of any other software, there are packages that greatly enhance its abilities, with many of the plugins functionality relying on them and are recommended for optimal results.

- Distorm3[73] - A powerful disassembler library for x86/AMD64 architecture
- Yara[74] - A malware identification and classification tool
- PyCrypto[75] - The Python cryptography toolkit
- PIL[76] – a Python imaging library
- OpenPyxl[77] – A Python library for reading and writing Microsoft Excel files
- Ujson[78] - A JSON parsing library for exporting to HTML files

Installation of the above additional open source software can be either achieved via their source code or for OS X and our case via the *pip*[79], a package management system similar to *homebrew* but dedicated to software packages written in Python. Searching a package via *pip* is as simple as the following:

```
pip search ujson
```

resulting to:

```
ujson             - Ultra fast JSON encoder and decoder for Python
ujson_delta       - A diff/patch pair for JSON-serialized data structures.
SudokuJson2Pdf    - This project provides a tool which convert json file of
            sudokuinformation to pdf.
drf_ujson         - Django Rest Framework UJSON Renderer
ujsoncompare      - Json comparison tool
```

Then, to install the desired package like *ujson* in our case:

73 https://github.com/gdabah/distorm
74 https://plusvic.github.io/yara/
75 https://www.dlitz.net/software/pycrypto/
76 http://www.pythonware.com/products/pil/
77 https://pypi.python.org/pypi/openpyxl
78 https://pypi.python.org/pypi/ujson
79 https://pip.pypa.io/en/stable/

pip install ujson

After completing the installation of all the recommended packages, the Volatility Framework is armed to the max for analysing memory dumps.

3.7.1 Memory Dumps

Putting the Volatility Framework into practice requires the memory files that are to be analysed. Acquiring these files in some cases may not be a trivial task and may require a series of considerations and steps to be taken in order to preserve the evidence a forensic analyst is looking for. Following is a decision tree diagram for dealing with memory acquisition and the decision one has to make, made by the creators of the framework [18]:

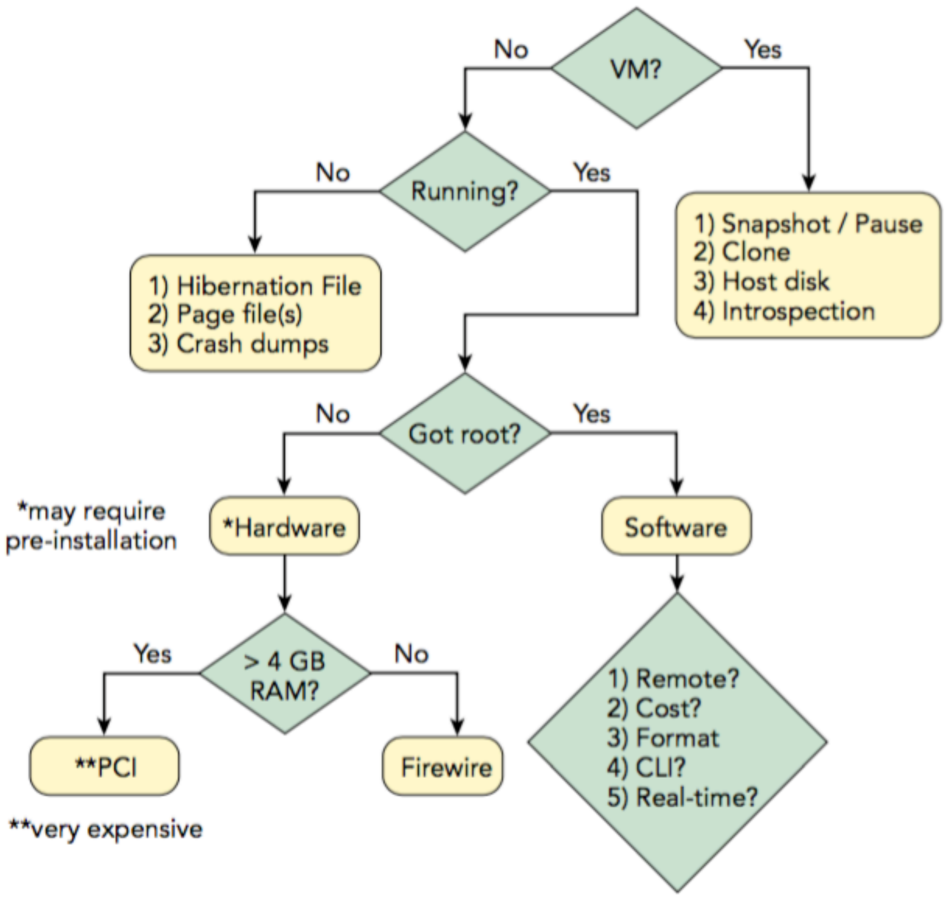


Figure 3-19, Memory acquisition decision tree diagram

Apart from these factors mentioned in Figure 3-19, there are some more, equivalently important to be taken under consideration for memory acquisition:

- *Cost*. Hardware solutions like CaptureGUARD[80] or EnCase[81] require thousands of US dollars but they have the advantage of being platform agnostic, as Carrier and Grand point out in their paper[70]. Software solutions like the Moonsols Windows Memory Toolkit[82] may cost way less or even be free of charge like Belkasoft's Live RAM Capturer[83]. However, in this case memory acquisition on different platforms require a separate solution for each platform.
- *Timeline*. Memory most of the times volatile. A successful acquisition does not guarantee that the artifacts required by a forensic analyst will be included in the memory dump, especially when the acquisition has to be achieved on a live system. For example, when looking for the activity of a specific malware that communicates with its command and control server via UDP[84], one has to acquire the RAM of the system when there is a connection with said servers in order for the information to be present in the memory dump.

The Volatility Framework is a not a memory acquisition tool, nor does it contain on. However, it does support memory acquisition of up to 4 GB via Firewire for OS X and Linux, by taking advantage of a library[85]

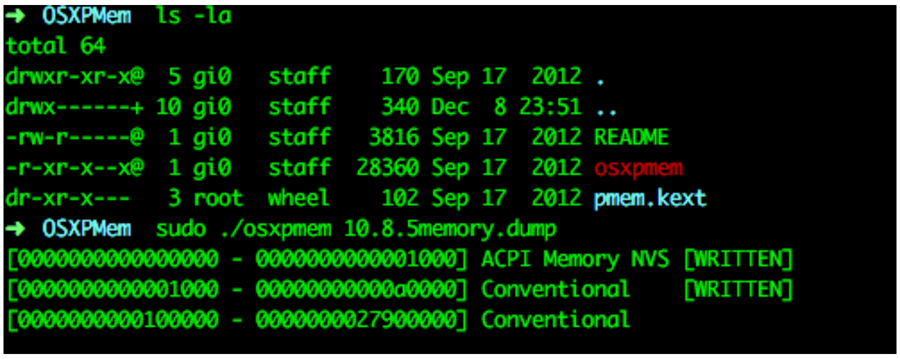
python vol.py –l firewire://forensic1394/<devno> plugin [options]

where *devno* is the number of the device attached via Firewire.

Following are some examples of software memory acquisition along with the tool used in each case.

[80] http://www.windowsscope.com/index.php?page=shop.product_details&flypage=flypage.tpl&product_id=29&category_id=3&option=com_virtuemart&Itemid=34
[81] https://www.guidancesoftware.com/
[82] http://www.moonsols.com/#pricing
[83] https://belkasoft.com/ram-capturer
[84] User Datagram Protocol, a transport layer protocol
[85] https://freddie.witherden.org/tools/libforensic1394/

For OS X, there is *OSXPMem* by Johannes Stuettgen[47] which works reliably for every major version of the operating system from 10.7 and onwards.



```
➜ OSXPMem  ls -la
total 64
drwxr-xr-x@  5 gi0   staff    170 Sep 17  2012 .
drwx------+ 10 gi0   staff    340 Dec  8 23:51 ..
-rw-r-----@  1 gi0   staff   3816 Sep 17  2012 README
-r-xr-x--x@  1 gi0   staff  28360 Sep 17  2012 osxpmem
dr-xr-x---   3 root  wheel    102 Sep 17  2012 pmem.kext
➜ OSXPMem  sudo ./osxpmem 10.8.5memory.dump
[0000000000000000 - 0000000000001000] ACPI Memory NVS [WRITTEN]
[0000000000001000 - 00000000000a0000] Conventional    [WRITTEN]
[0000000000100000 - 0000000027900000] Conventional
```

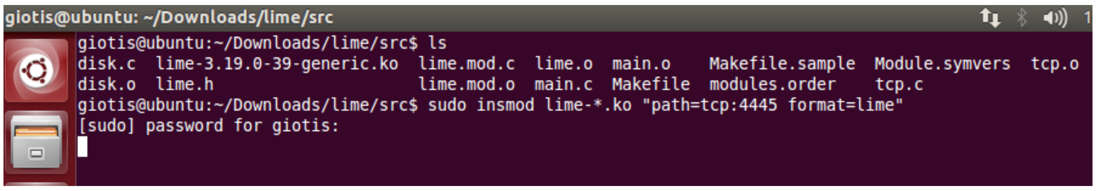
Figure 3-20, OSXPMem directory contents and memory acquisition

OSXPMem works without a problem for the x64 architecture but according to its developer it can also be compiled to work for the x86 architecture.

For the Linux platform an open source solution for memory acquisition is Lime[86] a loadable kernel module, introduced in 2012 by Joe Sylve[48]. To use Lime, we have to build the module via *make*[87] for the particular system which the memory acquisition will take place. Upon creating the module, we load it into the kernel with *insmod*[88]*:*

sudo insmod lime-3.19.0-39-generic.ko "path=tcp:4445 format=lime"

This way the operating system loads the module into the kernel and awaits for a TCP connection on port 4445 in order to send the contents of RAM to the remote system, in *lime* format.



```
giotis@ubuntu: ~/Downloads/lime/src
giotis@ubuntu:~/Downloads/lime/src$ ls
disk.c  lime-3.19.0-39-generic.ko  lime.mod.c  lime.o  main.o    Makefile.sample  Module.symvers  tcp.o
disk.o  lime.h                     lime.mod.o  main.c  Makefile  modules.order    tcp.c
giotis@ubuntu:~/Downloads/lime/src$ sudo insmod lime-*.ko "path=tcp:4445 format=lime"
[sudo] password for giotis:
```

Figure 3-21, Lime module loaded into the Linux kernel

[86] https://github.com/504ensicslabs/lime
[87] https://www.gnu.org/software/make/
[88] http://linux.die.net/man/8/insmod

To acquire the memory dump we connect to the system with netcat[89] and dump the contents into a file

nc 192.168.169.131 > ubuntu.lime

where 192.168.169.131 the IP of the target system and Ubuntu.lime the content of its memory. Since Ubuntu has most of ports closed by the kernel firewall, in order to connect to the target system we are required to open port 4445 using *iptables*[90], a utility for configuring the firewall:

sudo iptables -I INPUT -p tcp --dport 4445 -j ACCEPT

Using the above command, we configure the firewall to allow by default input connection at port 4445 using the TCP protocol.

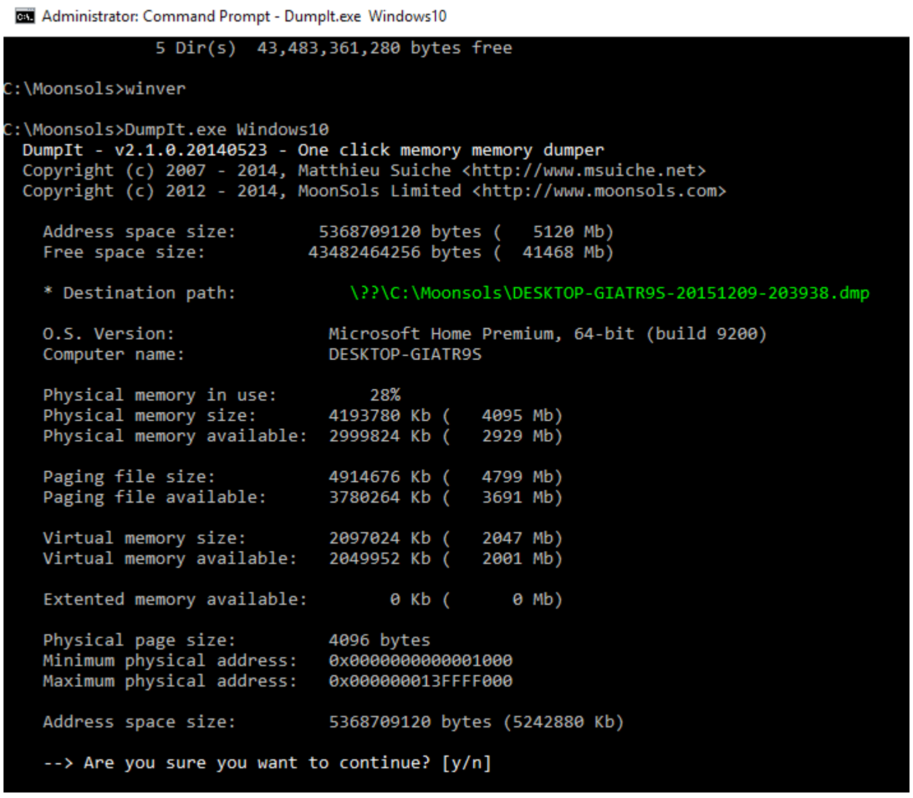
In Windows platforms there is a plethora of software solutions for memory acquisition. We used the Moonsols Memory Acquisition Tool mentioned earlier. More specifically, the edition we used is the Consultant Edition which on the contrary to the Free edition, supports both Microsoft Windows x86 and x64 architectures[46]. The toolkit contains five executables:

- *DumpIt.exe* that works for Microsoft Windows XP, 2003, 2008, Vista, 2008 R2, 7, 8 32-bits and 64-bits (x64) Edition. *DumpIt* can be used with scripts or/and batch files. It also provides an interactive mode for the acquisition of memory.
- *Hibr2dmp.exe* and *hibr2bin.exe* which works with Microsoft Windows XP, 2003, 2008, Vista, 2008 R2, 7, 8 32-bits and 64-bits (x64) Microsoft Windows hibernation files, including corrupted hibernation files. The produced file is hashed with the md5[91] algorithm.
- *Dmp2bin.exe* which works with Microsoft Windows XP, 2003, 2008, Vista, 2008 R2, 7, 8 32-bits and 64-bits (x64) Microsoft full memory crash dump files. The produced file is hashed with the md5 algorithm.

[89] http://linux.die.net/man/1/nc
[90] http://linux.die.net/man/8/iptables
[91] https://www.ietf.org/rfc/rfc1321.txt

- *Bin2dmp.exe* that works with Microsoft Windows XP, 2003, 2008, Vista, 2008 R2, 7, 8 32-bits and 64-bits (x64) raw memory snapshots (windd[92], VMWare[93]). The produced file is hashed with the md5 algorithm.

Although the Toolkit does not officially support the latest version of Microsoft's operating system, it works as expected (Figure 3-22)



```
Administrator: Command Prompt - DumpIt.exe  Windows10
               5 Dir(s)  43,483,361,280 bytes free

C:\Moonsols>winver

C:\Moonsols>DumpIt.exe Windows10
  DumpIt - v2.1.0.20140523 - One click memory memory dumper
  Copyright (c) 2007 - 2014, Matthieu Suiche <http://www.msuiche.net>
  Copyright (c) 2012 - 2014, MoonSols Limited <http://www.moonsols.com>

    Address space size:        5368709120 bytes (   5120 Mb)
    Free space size:          43482464256 bytes (  41468 Mb)

    * Destination path:           \??\C:\Moonsols\DESKTOP-GIATR9S-20151209-203938.dmp

    O.S. Version:               Microsoft Home Premium, 64-bit (build 9200)
    Computer name:              DESKTOP-GIATR9S

    Physical memory in use:         28%
    Physical memory size:       4193780 Kb (   4095 Mb)
    Physical memory available:  2999824 Kb (   2929 Mb)

    Paging file size:           4914676 Kb (   4799 Mb)
    Paging file available:      3780264 Kb (   3691 Mb)

    Virtual memory size:        2097024 Kb (   2047 Mb)
    Virtual memory available:   2049952 Kb (   2001 Mb)

    Extented memory available:        0 Kb (      0 Mb)

    Physical page size:         4096 bytes
    Minimum physical address:   0x0000000000001000
    Maximum physical address:   0x000000013FFFF000

    Address space size:         5368709120 bytes (5242880 Kb)

    --> Are you sure you want to continue? [y/n]
```

Figure 3-22, Acquiring the memory of a Windows 10 x64 system

As stated earlier, the Volatility Framework can analyze a series of file formats from RAW and *lime* files to Windows hibernation files and memory snapshots originating from virtualization software like Oracle's Virtualbox[94] and VMWare's Workstation[95].

Following are two cumulative tables of every memory image and file format that are supported by the Volatility Framework (Tables 3-6 and 3-7)

[92] A deprecated tool for memory dumping, http://forensicswiki.org/wiki/WinDD
[93] https://www.vmware.com/
[94] https://www.virtualbox.org/
[95] https://www.vmware.com/products/workstation

Supported memory images	
64-bit Windows Server 2012 and 2012 R2	32- and 64-bit Windows XP (SP2 and SP3)
32- and 64-bit Windows 10 (initial/basic support)	32- and 64-bit Linux kernels from 2.6.11 to 4.2.3
32- and 64-bit Windows 8, 8.1, and 8.1 Update 1	32-bit 10.5.x Leopard
32- and 64-bit Windows 7 (all service packs)	32- and 64-bit 10.6.x Snow Leopard
32- and 64-bit Windows Server 2008 (all service packs)	32- and 64-bit 10.7.x Lion
64-bit Windows Server 2008 R2 (all service packs)	64-bit 10.8.x Mountain Lion
32- and 64-bit Windows Vista (all service packs)	64-bit 10.9.x Mavericks
32- and 64-bit Windows Server 2003 (all service packs)	64-bit 10.10.x Yosemite
	64-bit 10.11.x El Capitan

Table 3-6, Supported memory images

Supported file formats	
Firewire (IEEE 1394)	32- and 64-bit MachO files
Raw/Padded Physical Memory	Virtualbox Core Dumps
Expert Witness (EWF)	VMware Saved State (.vmss) and Snapshot (.vmsn)
32- and 64-bit Windows Crash Dump	HPAK Format (FastDump)
32- and 64-bit Windows Hibernation	QEMU memory dumps

Table 3-7, Supported file formats

For our research we implemented a series of virtual machines using VMWare's virtualization solution, Workstation and Fusion[96], whose memory images formats, *.vmsn (*snapshots) and *.vmss* (saved states), can be analysed thanks to an open source python parser, *vmsnparse*[97]. It should be noted that there is one more related memory format, *.vmem*, which is similar to the RAW format.

[96] https://www.vmware.com/products/fusion
[97] https://code.google.com/p/vmsnparser/

4 Chapter 4 - Design and Development of a Volatility Framework Plugin

4.1 Setting up the test bed

Multiple Operating Systems are going to be tested in order to test and verify the functionality of the Volatility plugins and the respective malware samples.

4.1.1 Operating Systems and Platforms

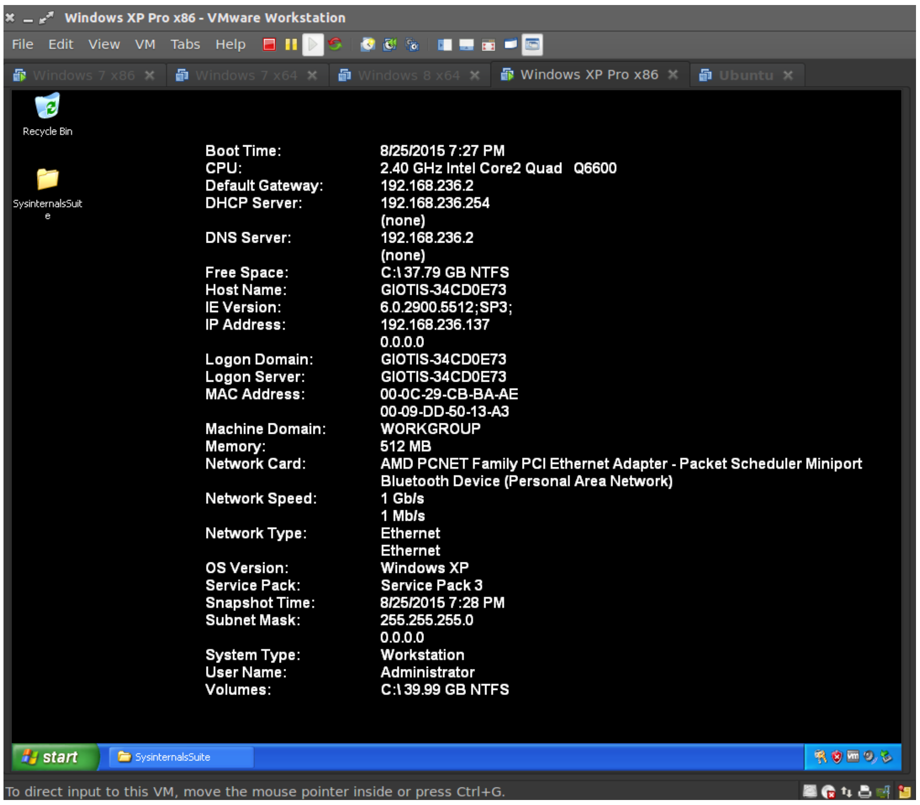
Multiple virtual machines have been created and a number of snapshots of their states were generated. The virtualization software of choice is VMWare Workstation Pro for Linux and VMWare Fusion Pro for OS X. The host operating systems are Ubuntu 15.04 x64 and OSX El Capitan, version 10.11.2. In total nine different virtual machines have been created. All are implemented for testing purposes and from their memory states samples are going to be acquired after infecting them with malware. For the analysis of the acquired samples, reverse engineering, coding and forensic analysis two physical systems are going to be used as test beds.

The virtual machine created and used for testing purposes are:

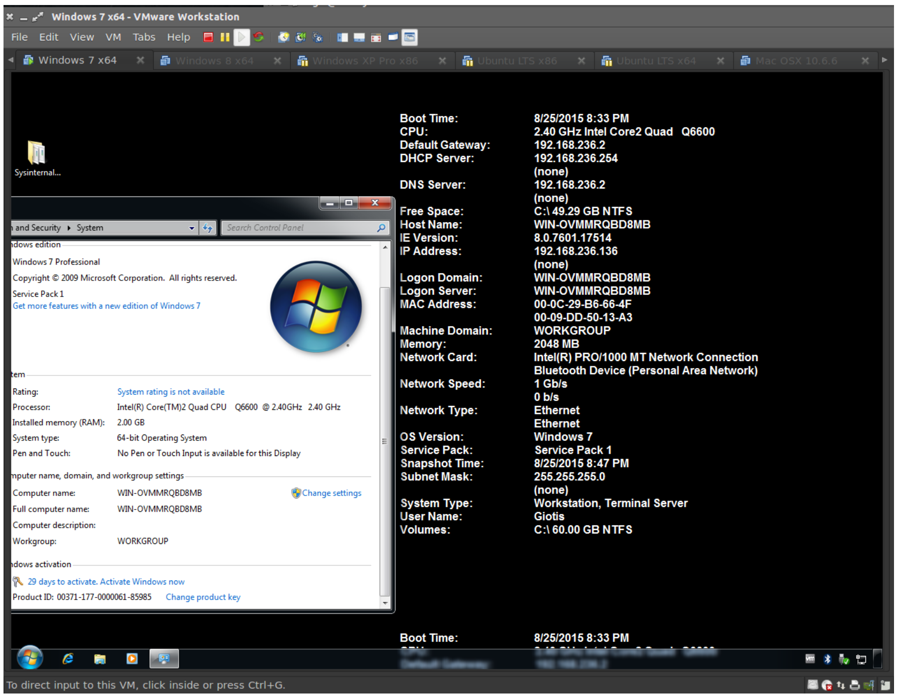
- Windows XP Pro SP3 x86
- Windows 7 Pro N SP1 x86
- Windows 7 Pro N SP1 x64
- Windows 8.1 x64
- Windows 10 Pro x64
- Ubuntu LTS 14.04.01 x86
- Ubuntu LTS 14.04.01 x64
- OS X Mountain Lion (10.8.5)

The host systems used for the analysis are based on Ubuntu Linux 15.04 x64 and OSX El Capitan, 10.11.2.

Following are pictures of said virtual machines, along with their hardware specifications:



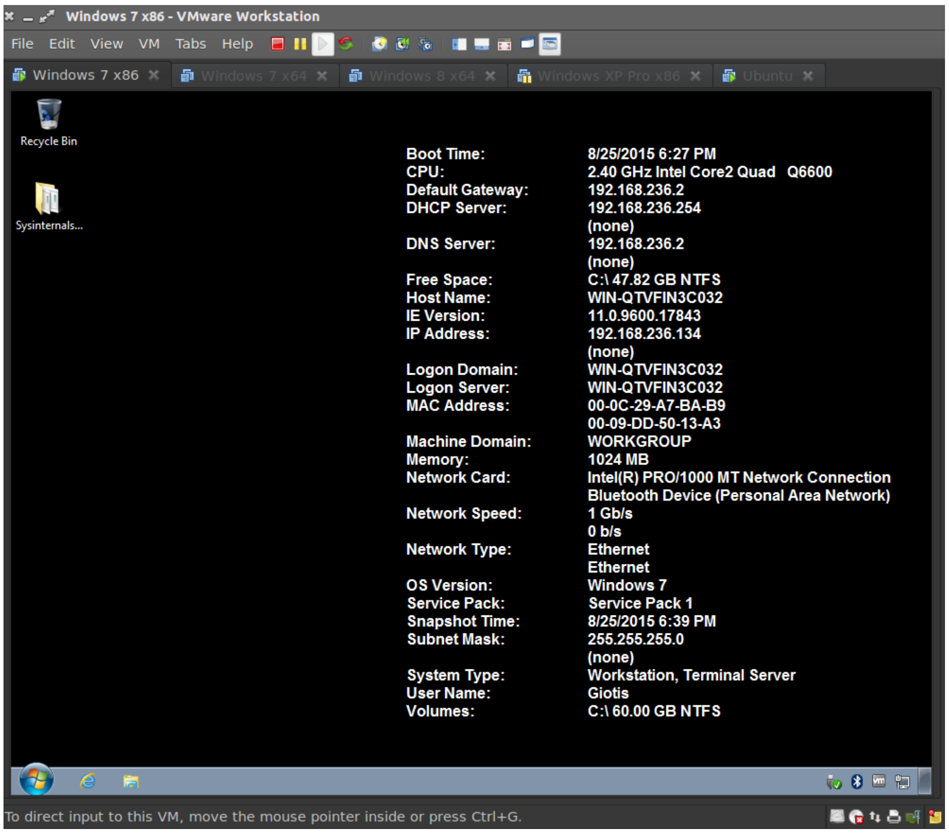
```
Boot Time:          8/25/2015 7:27 PM
CPU:                2.40 GHz Intel Core2 Quad   Q6600
Default Gateway:    192.168.236.2
DHCP Server:        192.168.236.254
                    (none)
DNS Server:         192.168.236.2
                    (none)
Free Space:         C:\ 37.79 GB NTFS
Host Name:          GIOTIS-34CD0E73
IE Version:         6.0.2900.5512;SP3;
IP Address:         192.168.236.137
                    0.0.0.0
Logon Domain:       GIOTIS-34CD0E73
Logon Server:       GIOTIS-34CD0E73
MAC Address:        00-0C-29-CB-BA-AE
                    00-09-DD-50-13-A3
Machine Domain:     WORKGROUP
Memory:             512 MB
Network Card:       AMD PCNET Family PCI Ethernet Adapter - Packet Scheduler Miniport
                    Bluetooth Device (Personal Area Network)
Network Speed:      1 Gb/s
                    1 Mb/s
Network Type:       Ethernet
                    Ethernet
OS Version:         Windows XP
Service Pack:       Service Pack 3
Snapshot Time:      8/25/2015 7:28 PM
Subnet Mask:        255.255.255.0
                    0.0.0.0
System Type:        Workstation
User Name:          Administrator
Volumes:            C:\ 39.99 GB NTFS
```



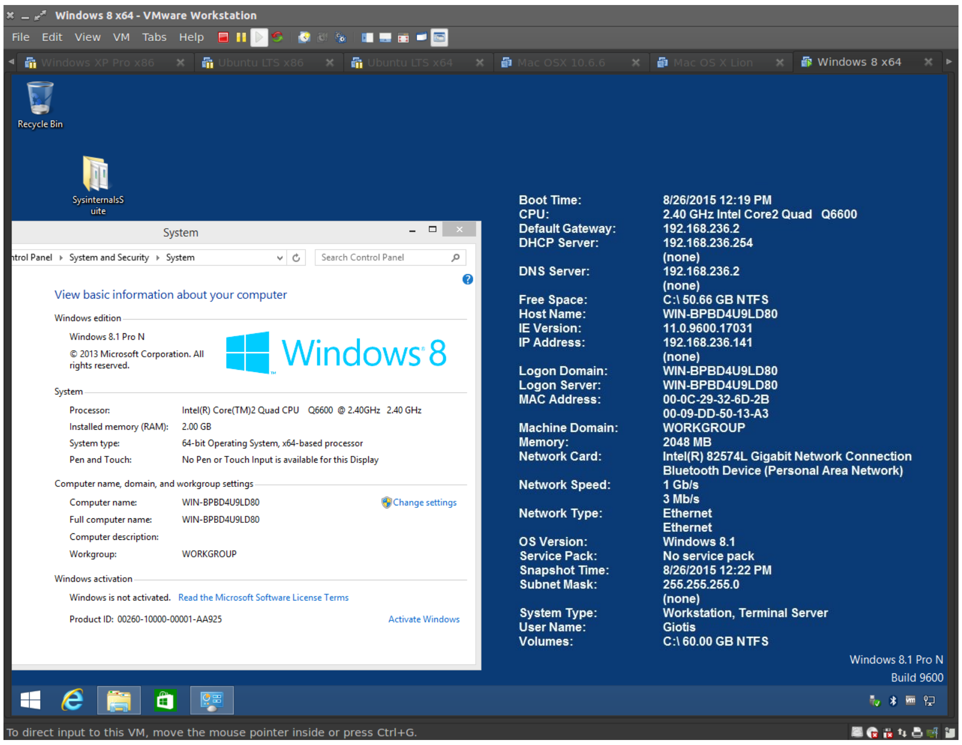
```
ndows edition
Windows 7 Professional
Copyright © 2009 Microsoft Corporation.  All rights reserved.
Service Pack 1
Get more features with a new edition of Windows 7

tem
Rating:                  System rating is not available
Processor:               Intel(R) Core(TM)2 Quad CPU    Q6600  @ 2.40GHz   2.40 GHz
Installed memory (RAM):  2.00 GB
System type:             64-bit Operating System
Pen and Touch:           No Pen or Touch Input is available for this Display

mputer name, domain, and workgroup settings
Computer name:           WIN-OVMMRQBD8MB            Change settings
Full computer name:      WIN-OVMMRQBD8MB
Computer description:
Workgroup:               WORKGROUP

ndows activation
29 days to activate. Activate Windows now
Product ID: 00371-177-0000061-85985     Change product key
```

```
Boot Time:          8/25/2015 8:33 PM
CPU:                2.40 GHz Intel Core2 Quad   Q6600
Default Gateway:    192.168.236.2
DHCP Server:        192.168.236.254
                    (none)
DNS Server:         192.168.236.2
                    (none)
Free Space:         C:\ 49.29 GB NTFS
Host Name:          WIN-OVMMRQBD8MB
IE Version:         8.0.7601.17514
IP Address:         192.168.236.136
                    (none)
Logon Domain:       WIN-OVMMRQBD8MB
Logon Server:       WIN-OVMMRQBD8MB
MAC Address:        00-0C-29-B6-66-4F
                    00-09-DD-50-13-A3
Machine Domain:     WORKGROUP
Memory:             2048 MB
Network Card:       Intel(R) PRO/1000 MT Network Connection
                    Bluetooth Device (Personal Area Network)
Network Speed:      1 Gb/s
                    0 b/s
Network Type:       Ethernet
                    Ethernet
OS Version:         Windows 7
Service Pack:       Service Pack 1
Snapshot Time:      8/25/2015 8:47 PM
Subnet Mask:        255.255.255.0
                    (none)
System Type:        Workstation, Terminal Server
User Name:          Giotis
Volumes:            C:\ 60.00 GB NTFS

Boot Time:          8/25/2015 8:33 PM
```

Figure 4-1, Windows XP and Windows 7 x64 VMs


Windows 7 x86 - VMware Workstation
File Edit View VM Tabs Help
Windows 7 x86
Windows 7 x64
Windows 8 x64
Windows XP Pro x86
Ubuntu
Recycle Bin
Sysinternals...
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

```
Boot Time:          8/25/2015 6:27 PM
CPU:                2.40 GHz Intel Core2 Quad   Q6600
Default Gateway:    192.168.236.2
DHCP Server:        192.168.236.254
                    (none)
DNS Server:         192.168.236.2
                    (none)
Free Space:         C:\ 47.82 GB NTFS
Host Name:          WIN-QTVFIN3C032
IE Version:         11.0.9600.17843
IP Address:         192.168.236.134
                    (none)
Logon Domain:       WIN-QTVFIN3C032
Logon Server:       WIN-QTVFIN3C032
MAC Address:        00-0C-29-A7-BA-B9
                    00-09-DD-50-13-A3
Machine Domain:     WORKGROUP
Memory:             1024 MB
Network Card:       Intel(R) PRO/1000 MT Network Connection
                    Bluetooth Device (Personal Area Network)
Network Speed:      1 Gb/s
                    0 b/s
Network Type:       Ethernet
                    Ethernet
OS Version:         Windows 7
Service Pack:       Service Pack 1
Snapshot Time:      8/25/2015 6:39 PM
Subnet Mask:        255.255.255.0
                    (none)
System Type:        Workstation, Terminal Server
User Name:          Giotis
Volumes:            C:\ 60.00 GB NTFS
```


Windows 8 x64 - VMware Workstation
File Edit View VM Tabs Help
Windows XP Pro x86
Ubuntu LTS x86
Ubuntu LTS x64
Mac OSX 10.6.6
Mac OS X Lion
Windows 8 x64
Recycle Bin
SysinternalsSuite
System
ntrol Panel › System and Security › System
Search Control Panel
Windows 8
Change settings
Read the Microsoft Software License Terms
Activate Windows
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

View basic information about your computer

Windows edition

Windows 8.1 Pro N

© 2013 Microsoft Corporation. All rights reserved.

System

Processor:	Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz 2.40 GHz
Installed memory (RAM):	2.00 GB
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

Computer name, domain, and workgroup settings

Computer name:	WIN-BPBD4U9LD80
Full computer name:	WIN-BPBD4U9LD80
Computer description:	
Workgroup:	WORKGROUP

Windows activation

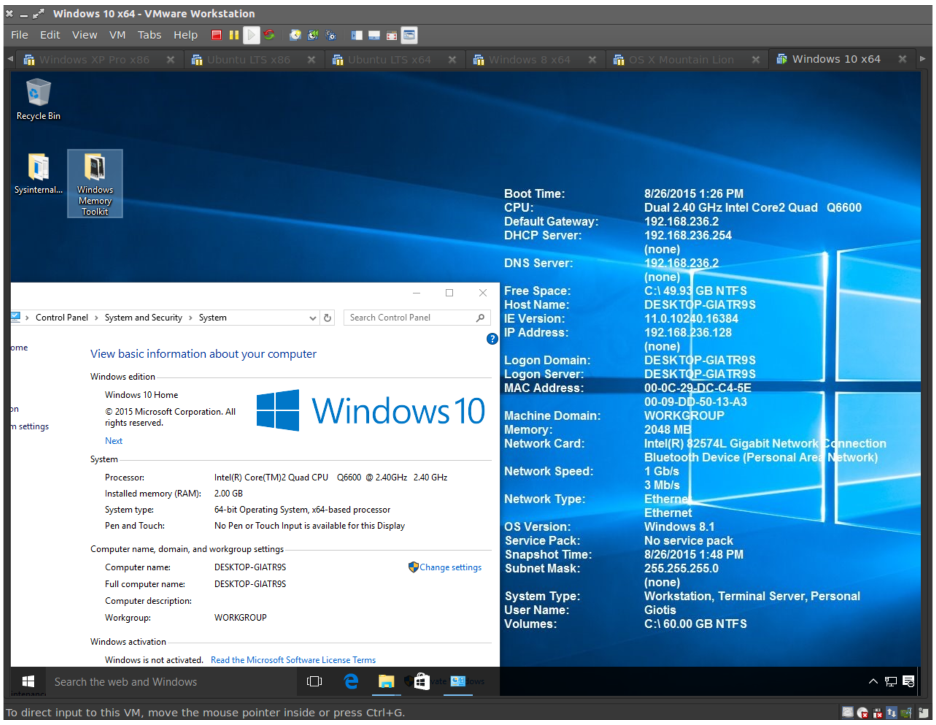
Windows is not activated.

Product ID: 00260-10000-00001-AA925

```
Boot Time:          8/26/2015 12:19 PM
CPU:                2.40 GHz Intel Core2 Quad   Q6600
Default Gateway:    192.168.236.2
DHCP Server:        192.168.236.254
                    (none)
DNS Server:         192.168.236.2
                    (none)
Free Space:         C:\ 50.66 GB NTFS
Host Name:          WIN-BPBD4U9LD80
IE Version:         11.0.9600.17031
IP Address:         192.168.236.141
                    (none)
Logon Domain:       WIN-BPBD4U9LD80
Logon Server:       WIN-BPBD4U9LD80
MAC Address:        00-0C-29-32-6D-2B
                    00-09-DD-50-13-A3
Machine Domain:     WORKGROUP
Memory:             2048 MB
Network Card:       Intel(R) 82574L Gigabit Network Connection
                    Bluetooth Device (Personal Area Network)
Network Speed:      1 Gb/s
                    3 Mb/s
Network Type:       Ethernet
                    Ethernet
OS Version:         Windows 8.1
Service Pack:       No service pack
Snapshot Time:      8/26/2015 12:22 PM
Subnet Mask:        255.255.255.0
                    (none)
System Type:        Workstation, Terminal Server
User Name:          Giotis
Volumes:            C:\ 60.00 GB NTFS
```

Windows 8.1 Pro N
Build 9600

Figure 4-2, Windows 7 x86 and Windows 8.1 x64 VMs


Windows 10 x64 - VMware Workstation
File Edit View VM Tabs Help
Windows XP Pro x86
Ubuntu LTS x86
Ubuntu LTS x64
Windows 8 x64
OS X Mountain Lion
Windows 10 x64
Recycle Bin
Sysinternal...
Windows Memory Toolkit
Control Panel › System and Security › System
Search Control Panel
Search the web and Windows
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

View basic information about your computer

Windows edition
Windows 10 Home
© 2015 Microsoft Corporation. All rights reserved.
Next

System

Processor:	Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz 2.40 GHz
Installed memory (RAM):	2.00 GB
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

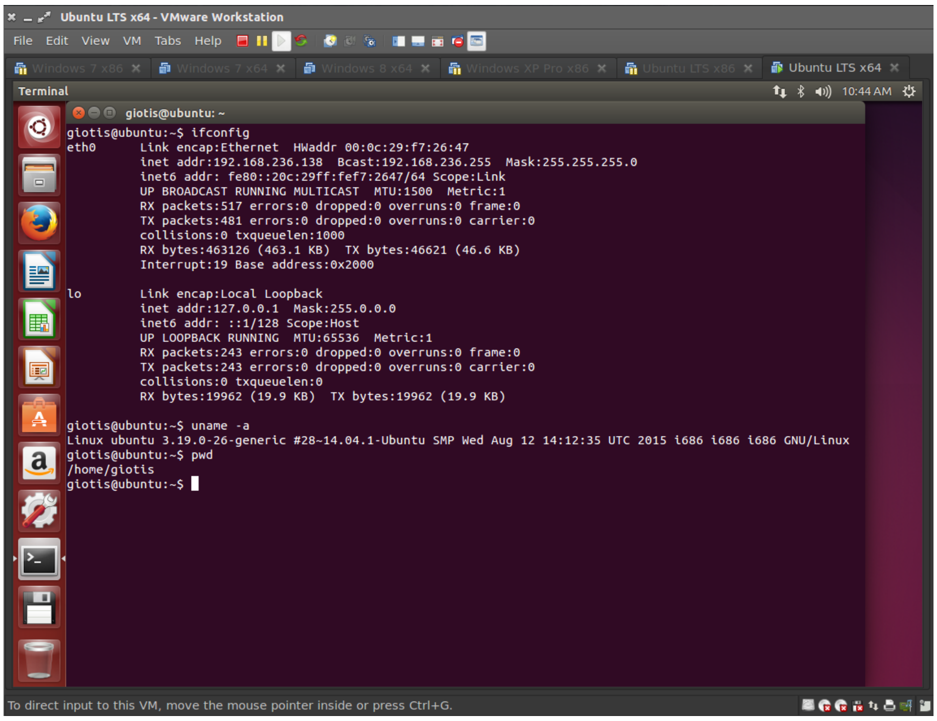
Computer name, domain, and workgroup settings

Computer name:	DESKTOP-GIATR9S
Full computer name:	DESKTOP-GIATR9S
Computer description:	
Workgroup:	WORKGROUP

Change settings

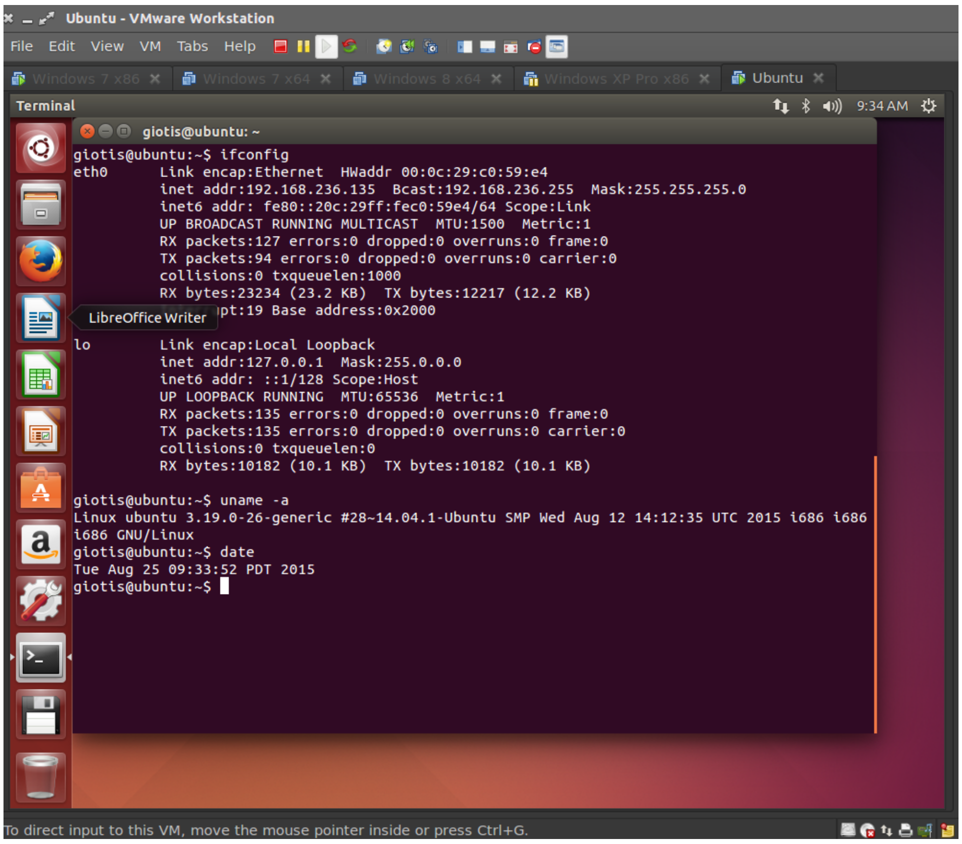
Windows activation
Windows is not activated. Read the Microsoft Software License Terms

```
Boot Time:          8/26/2015 1:26 PM
CPU:                Dual 2.40 GHz Intel Core2 Quad  Q6600
Default Gateway:    192.168.236.2
DHCP Server:        192.168.236.254
                    (none)
DNS Server:         192.168.236.2
                    (none)
Free Space:         C:\ 49.93 GB NTFS
Host Name:          DESKTOP-GIATR9S
IE Version:         11.0.10240.16384
IP Address:         192.168.236.128
                    (none)
Logon Domain:       DESKTOP-GIATR9S
Logon Server:       DESKTOP-GIATR9S
MAC Address:        00-0C-29-DC-C4-5E
                    00-09-DD-50-13-A3
Machine Domain:     WORKGROUP
Memory:             2048 MB
Network Card:       Intel(R) 82574L Gigabit Network Connection
                    Bluetooth Device (Personal Area Network)
Network Speed:      1 Gb/s
                    3 Mb/s
Network Type:       Ethernet
                    Ethernet
OS Version:         Windows 8.1
Service Pack:       No service pack
Snapshot Time:      8/26/2015 1:48 PM
Subnet Mask:        255.255.255.0
                    (none)
System Type:        Workstation, Terminal Server, Personal
User Name:          Giotis
Volumes:            C:\ 60.00 GB NTFS
```

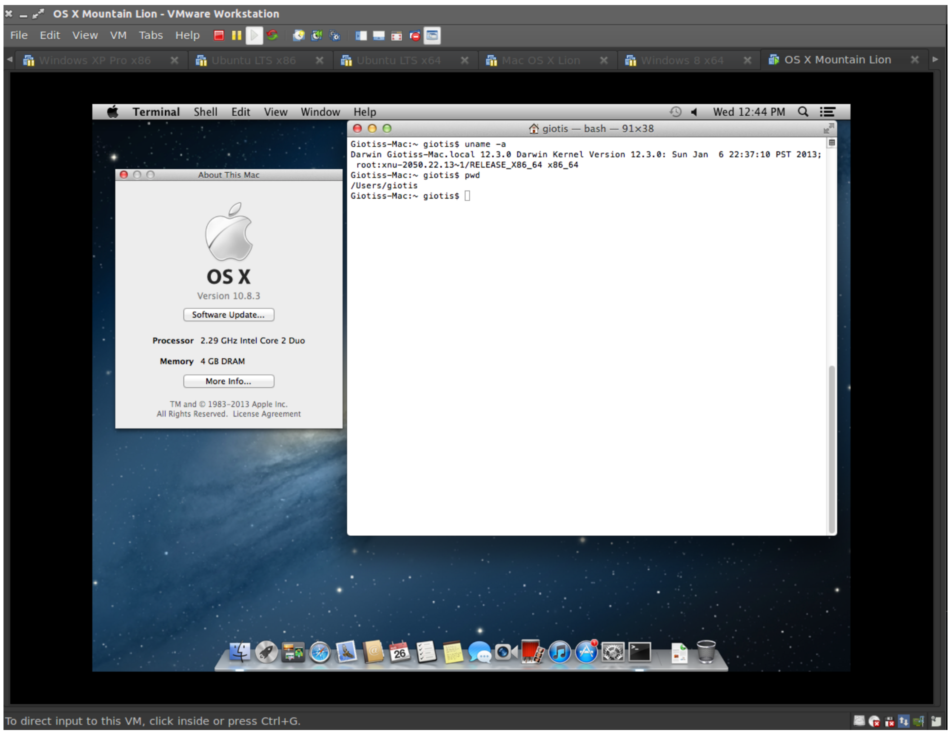

Ubuntu LTS x64 - VMware Workstation
File Edit View VM Tabs Help
Windows 7 x86
Windows 7 x64
Windows 8 x64
Windows XP Pro x86
Ubuntu LTS x86
Ubuntu LTS x64
Terminal
10:44 AM
giotis@ubuntu: ~
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

```
giotis@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:f7:26:47
          inet addr:192.168.236.138  Bcast:192.168.236.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fef7:2647/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:517 errors:0 dropped:0 overruns:0 frame:0
          TX packets:481 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:463126 (463.1 KB)  TX bytes:46621 (46.6 KB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:243 errors:0 dropped:0 overruns:0 frame:0
          TX packets:243 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:19962 (19.9 KB)  TX bytes:19962 (19.9 KB)

giotis@ubuntu:~$ uname -a
Linux ubuntu 3.19.0-26-generic #28~14.04.1-Ubuntu SMP Wed Aug 12 14:12:35 UTC 2015 i686 i686 i686 GNU/Linux
giotis@ubuntu:~$ pwd
/home/giotis
giotis@ubuntu:~$
```

Figure 4-3, Windows 10 and Ubuntu LTS 15.04 x64 VMs


Ubuntu - VMware Workstation
File Edit View VM Tabs Help
Windows 7 x86
Windows 7 x64
Windows 8 x64
Windows XP Pro x86
Ubuntu
Terminal
9:34 AM
giotis@ubuntu: ~
LibreOffice Writer
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

```
giotis@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:c0:59:e4
          inet addr:192.168.236.135  Bcast:192.168.236.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fec0:59e4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:127 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:23234 (23.2 KB)  TX bytes:12217 (12.2 KB)
                 pt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:135 errors:0 dropped:0 overruns:0 frame:0
          TX packets:135 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:10182 (10.1 KB)  TX bytes:10182 (10.1 KB)

giotis@ubuntu:~$ uname -a
Linux ubuntu 3.19.0-26-generic #28~14.04.1-Ubuntu SMP Wed Aug 12 14:12:35 UTC 2015 i686 i686
i686 GNU/Linux
giotis@ubuntu:~$ date
Tue Aug 25 09:33:52 PDT 2015
giotis@ubuntu:~$
```


OS X Mountain Lion - VMware Workstation
File Edit View VM Tabs Help
Windows XP Pro x86
Ubuntu LTS x86
Ubuntu LTS x64
Mac OS X Lion
Windows 8 x64
OS X Mountain Lion
Terminal Shell Edit View Window Help
Wed 12:44 PM
giotis — bash — 91×38
About This Mac
Software Update...
More Info...
TM and © 1983–2013 Apple Inc.
All Rights Reserved. License Agreement
To direct input to this VM, click inside or press Ctrl+G.

```
Giotiss-Mac:~ giotis$ uname -a
Darwin Giotiss-Mac.local 12.3.0 Darwin Kernel Version 12.3.0: Sun Jan  6 22:37:10 PST 2013;
 root:xnu-2050.22.13~1/RELEASE_X86_64 x86_64
Giotiss-Mac:~ giotis$ pwd
/Users/giotis
Giotiss-Mac:~ giotis$
```

OS X
Version 10.8.3

Processor 2.29 GHz Intel Core 2 Duo

Memory 4 GB DRAM

Figure 4-4, Ubuntu LTS 15.04 x86 and OS X Mountain Lion VMs

4.1.2 Hardware Requirements

Since virtualization software is the primary mean of research, hardware requirements of the test-bed must meat those of the software we are using, VMWare Workstation Pro and VMWare Fusion Pro. More specifically, for Workstation Pro hardware requirements[98] are a standard x86-PC with Intel and AMD processor. VMware recommends the following:

- 64-bit x86 Intel Core Duo Processor or equivalent, AMD Athlon 64 FX Dual Core Processor or equivalent
- 1.3GHz speed or faster
- 2GB RAM minimum, 4GB RAM and above recommended

Workstation Pro installation requires:

1.2 GB of available disk space for the application. Additional hard disk space required for each virtual machine, depending on the minimum requirements of the operating system it hosts.

For Windows DirectX 10 support in a virtual machine:

- 3GB RAM (Host PC)
- Intel Dual Core, 2.2GHz and above or AMD Athlon 4200+ and above
- NVIDIA GeForce 8800GT and above or ATI Radeon HD 2600 and above

For VMWare Fusion Pro the hardware requirements are the following[99]:

- Any 64--bit capable Intel® Mac, compatible with Core 2 Duo, Xeon, i3, i5, i7 processors or better
- Minimum 4GB of RAM
- 750MB free disk space for VMware Fusion and at least 5GB for each virtual machine, depending on the client's operating system
- Mac OS X 10.9.0 or later
- Operating system installation media (disk or disk image) for virtual machines.

[98] https://www.vmware.com/products/workstation/faqs/install-requirements
[99] https://www.vmware.com/support/fusion/faq/requirements

The recommended graphics hardware for Windows DirectX 10 or OpenGL 3.3 support according to VMWare is equivalent to:

- NVIDIA 8600M or better
- ATI 2600 or better

Regarding the hardware specifications of the clients, all virtual machines meat the minimum requirements specified by the developers of each operating system. Moreover, due to antimalware techniques[71] [72] introduced to various modern malware, some hardware features like the size of the hard disk are purposely set to values well beyond the requirements of the client operating system. One of the advantages of virtualization is the ability to change the hardware settings of the operating system on the fly or after a reboot, which in some cases, like the analysis of a malware anti-debugging and anti-virtualization techniques similar to the *Tinba*[73], is a valuable feature.

4.1.3 Software Requirements

The client machines are default installations of the operating systems, updated to the latest version via their respective update mechanisms. In each case the appropriate software solution for memory acquisition was installed[100], so as to have the option of acquire the memory natively from inside the client, apart from the *.vmss* files prvided from VMWare's software.

For the host machines, Volatility Framework along with all the recommended packages was installed[101]. VMWare Workstation and Fusion as stated earlier were installed on both computers. Both products support all versions of Linux and Windows for client systems, however OS X is not supported by default. In order to run Apple's operating system via VMWare Workstation or Fusion, we patched the them via an open source tool written in Python, *Unlocker*[102].

[100] See Chapter 3, *Memory Dumps*
[101] See Chapter 3, *Installation and setting up a test environment*
[102] http://www.insanelymac.com/forum/files/file/339-unlocker/

For research purposes, three samples of DevilRobber[103], a Bitcoin malware for OSX, have been acquired thanks to Mikko Hypponen[104], CRO of F-Secure and are used to infected the corresponding virtual machine. The MD5 hashes of the samples:

MD5 (mdsa.1) = 16e3bc0415056eb15b2752613970b79d

MD5 (mdsa.2) = a123cc209802aa37cc62d23682e8cf8d

MD5 (mdsa.3) = 9e22ceb19f397f8cd018f90c857ab638

In order to test the functionality of a Volatility Framework plugin related to Bicoin, developed by Cem Gurkok[45] (see Appendix A), *MultiBit HD*[105], a Bitcoin wallet was also installed. For the Windows platform two samples of *BadMiner*[106] malware were investigated for the same purposes. The MD5 hashes of the samples:

MD5 (BadMinerSample1.exe) = 0761d41068167c83047d06b89f497343

MD5 (badminerSample2) = 6defe9a02352ddc0dba8f9949602c1d7

For the Linux platform, two samples of a Linux malware that in 2014 was found to hav infected more than 31.000 devices that were running Linux, the *Darlloz* worm [74], were used. The MD5 hashes of the samples:

MD5 (Darlloz sample 1) = df70db2fa08985c237892e83861bed50

MD5 (Darlloz sample 2) = bd3870a568e1019797bff113c39ad3e0

[103] https://www.f-secure.com/v-descs/backdoor_osx_devilrobber_a.shtml
[104] https://mikko.hypponen.com/
[105] https://multibit.org/
[106] https://www.symantec.com/security_response/writeup.jsp?docid=2011-081115-5847-99

4.2 A Bitcoin plugin for the Volatility Framework

The Volatility Foundation every year holds a plugins contest, for everyone to participate, awarding the top three plugins that are judged by them for their intuition, creativity and usefulness. In 2014 Cem Gurkok's [45] plugin for finding Bitcoin addresses and keys of a specific Bitcoin desktop client, Multibit HD[107] received an honorary mention.

Upon examining said plugin source code (see Appendix A), we researched the possibility of expanding its features to more than locating artifacts of the aforementioned client. Firstly, we compiled a list of all the Bitcoin desktop clients for OSX, which are used worldwide and are endorsed by the top sites regarding Bitcoin in the world[108]. The list in no particular order:

- Hive
- Bither
- Copay
- Electrum
- mSiGNA
- ArmoryQt
- Multibit HD

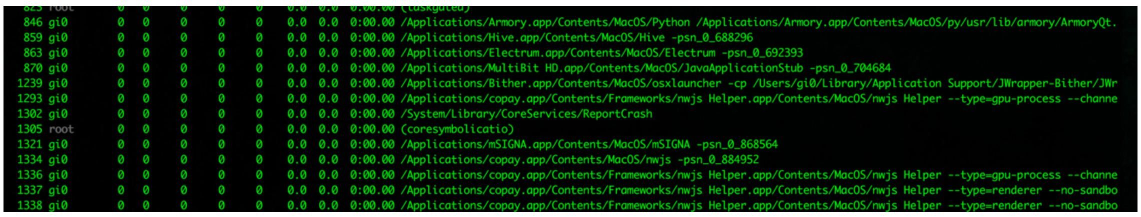
Then we proceeded installing each one of them and examine their footprint on three of the latest versions of OS X; Mountain Lion, Yosemite, El Capitan. In all cases we focused on unique attributes that would help pinpoint the existence of the clients on the system in order to enhance the plugin mentioned earlier for detecting an analyzing them. There are many possible ways of doing so. One of them is by using *Yara*[109], "a tool aimed at helping malware researchers to classify and identify malware samples". By using *Yara*, we can examine the executables and create rules that can uniquely identify them. A sample rule that contains information regarding its creator and the pattern of strings that is used, follows:

[107] https://multibit.org/
[108] http://www.alexa.com/topsites/category/Science/Social_Sciences/Economics/Financial_Economics/Currency_and_Money/Alternative_Monetary_Systems/Bitcoin
[109] https://plusvic.github.io/yara/

```
rule thesis_sample : sample
{
   meta:
      description = "This is an example"
      thread_level = 5
      in_the_wild = no

   strings:
      $a = {0A 00 00 30 90 00 00 8A 14 1D 90}
      $b = {0F 5Q D0 0F CC 11 90 00 2B 9E FF E7 00}
      $c = "TDGFOIHHNALMPQLOI"

   condition:
      $a or $b or $c
}
```

Another way of identifying unique properties in every client’s footprint is by examining the process listing. Every client has a unique process name tied to their actual name, making it an ideal way of locating their memory. More specifically:

- Hive is identified as *Hive*
- Bither is identified as *JWrapper-Bither*
- Multibit HD is identified as *JavaApplicationS*
- Copay is based on node.js[110] and should not be identified by the *nwjs* string alone. *Coppay.app* is used instead.
- Electrum as *Electrum*
- mSIGNA as *mSigna*
- ArmoryGt as *ArmoryQt*



```
 823 root     0   0    0    0    0   0.0  0.0  0:00.00 (taskgated)
 846 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/Armory.app/Contents/MacOS/Python /Applications/Armory.app/Contents/MacOS/py/usr/lib/armory/ArmoryQt.
 859 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/Hive.app/Contents/MacOS/Hive -psn_0_688296
 863 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/Electrum.app/Contents/MacOS/Electrum -psn_0_692393
 870 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/MultiBit HD.app/Contents/MacOS/JavaApplicationStub -psn_0_704684
1239 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/Bither.app/Contents/MacOS/osxlauncher -cp /Users/gi0/Library/Application Support/JWrapper-Bither/JWr
1293 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/copay.app/Contents/Frameworks/nwjs Helper.app/Contents/MacOS/nwjs Helper --type=gpu-process --channe
1302 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /System/Library/CoreServices/ReportCrash
1305 root     0   0    0    0    0   0.0  0.0  0:00.00 (coresymbolicatio)
1321 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/mSIGNA.app/Contents/MacOS/mSIGNA -psn_0_868564
1334 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/copay.app/Contents/MacOS/nwjs -psn_0_884952
1336 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/copay.app/Contents/Frameworks/nwjs Helper.app/Contents/MacOS/nwjs Helper --type=gpu-process --channe
1337 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/copay.app/Contents/Frameworks/nwjs Helper.app/Contents/MacOS/nwjs Helper --type=renderer --no-sandbo
1338 gi0      0   0    0    0    0   0.0  0.0  0:00.00 /Applications/copay.app/Contents/Frameworks/nwjs Helper.app/Contents/MacOS/nwjs Helper --type=renderer --no-sandbo
```

Figure 4-5, All clients running and their identifiers

[110] https://github.com/nwjs/nw.js

In addition, we analyzed samples of three OS X malware families that can lead to backdooring a system and illegal Bitcoin mining, *DevilRober*[111], *Kitmos*[112] and *Morcut*[113]. Since their processes and executables are on purpose randomly created in order to avoid such a similar kind of detection, *Yara* was used to create rules that will help locate them.

Creating a *Yara* rule can be achieved either by using the Python command line tool *yaragenerator*[114]*:*

```
python yaraGenerator.py ../OSX/ -r DevilRobber -a "Giotis" -d "DevilRobber sample" -
t "OSX" -f "exe"
```

or by using an automated service like *yaragenerator.com.* It is also possible to create them manually after reading the documentation, however the two solution mentioned are far more efficient when dealing with multiple samples.

[111]https://www.f-secure.com/v-descs/backdoor_osx_devilrobber_a.shtml
[112]https://www.symantec.com/security_response/writeup.jsp?docid=2013-051616-5911-99
[113]https://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/OSX~Morcut-A.aspx
[114]https://github.com/Xen0ph0n/YaraGenerator

5 Chapter 5 - Evaluation of the proposed plugin

In this chapter we are discussing the enhancements made to the Bitcoin plugin, along with the advantages and disadvantages they introduce.

5.1 Analysis of the code

Following is a snippet of code with the enhancements introduced (see Appendix A).

```
    def calculate(self):
        all_tasks = pstasks.mac_tasks(self._config).allprocs()
        bit_tasks = []
        btcclients = ['Electrum', 'Hive', 'JavaApplicationS', 'JWrapper-Bither',
'Coppay.app', 'mSigna', 'ArmoryQt']

        try:
          if self._config.PID:
            pidlist = [int(p) for p in self._config.PID.split(',')]
            bit_tasks = [t for t in all_tasks if t.p_pid in pidlist]
          else:
            i = 0
            while i<len(btcclients):
              client2check=btcclients[i]
              name_re = re.compile(client2check, re.I)
              bit_tasks = [t for t in all_tasks if name_re.search(str(t.p_comm))]
              i=i+1
        except:
          pass

        if len(bit_tasks) == 0:
          yield (None, None)
```

The framework upon request will search the memory sample for one or all of the clients introduced in the list btcclients and check their memory space for structures indicating the presence of a Bitcoin wallet or key.

For the malware samples referred earlier, a number of Yara rules where created for embedding into the plugin, that will help detecting their existence in a memory sample.

DevilRobber

```
rule DevilRobber : OSX
{
meta:
   author = "Giotis"
   date = "2015-11-10"
   description = "OSX"
   hash0 = "a123cc209802aa37cc62d23682e8cf8d"
   hash1 = "16e3bc0415056eb15b2752613970b79d"
   hash2 = "9e22ceb19f397f8cd018f90c857ab638"
   sample_filetype = "exe"
   yaragenerator = "https://github.com/Xen0ph0n/YaraGenerator"
strings:
   $string0 = "_read$UNIX2003"
   $string1 = "content-length"
   $string2 = "URLBase"
   $string3 = "/usr/lib/dyld"
   $string4 = "[FF02::C]"
   $string5 = "_listen$UNIX2003"
   $string6 = "_curl_global_init"
   $string7 = "_close$UNIX2003"
   $string8 = "_malloc"
   $string9 = "__const"
   $string10 = "rm -f s.txt"
   $string11 = "_inet_addr"
   $string12 = "presentationURL"
   $string13 = "__common"
   $string14 = "_recv$UNIX2003"
   $string15 = "_curl_easy_init"
   $string16 = "__DefaultRuneLocale"
   $string17 = "_freeaddrinfo"
   $string18 = "_strcmp"
condition:
   18 of them
}
```

Kitmos

```
rule Kitmos : OSX
{
meta:
    author = "Giotis"
    date = "2015-11-10"
    description = "Kitmos"
    hash0 = "39faa22eb9d6b750ec345efcb38189f5"
    hash1 = "d43dec59fa8e6629ff46ae9e56f698d8"
    hash2 = "2e5345da904bba1c116b818fc9d5ab8f"
    hash3 = "b3d49091875de190f200110c2f2032d4"
    hash4 = "f9fabd1637d190e0e0a5c117c71921fc"
    sample_filetype = "exe"
    yaragenerator = "https://github.com/Xen0ph0n/YaraGenerator"
strings:
    $string0 = "m_nLoadedCount"
    $string1 = " before "
    $string2 = "addObject:"
    $string3 = "/bin/sh"
    $string4 = "m_ExtArray"
    $string5 = "requestWithURL:"
    $string6 = "m_nSavedMoment"
    $string7 = "copyItemAtPath:toPath:error:"
    $string8 = "N37CXSRXLD110/"
    $string9 = "__TEXT"
    $string10 = "121207052050Z"
    $string11 = "removeAllObjects"
    $string12 = "N37CXSRXLD1"
    $string13 = "/System/Library/Frameworks/AppKit.framework/Ver-
sions/C/AppKit"
    $string14 = "dictionaryWithObjectsAndKeys:"
    $string15 = "TaskWrapperController"
    $string16 = "NSWindow"
    $string17 = "$Developer ID Certification Authority1"
    $string18 = "/usr/lib/libobjc.A.dylib"
condition:
    18 of them
}
```

Morcut

```
rule Morcut : OSX
{
meta:
   author = "Giotis"
   date = "2015-11-10"
   description = "Morcut"
   hash0 = "2c684cad7e75f17a57b6a6a1ca7198f3"
   hash1 = "acec5f00057d3ec94849511f3eddcb91"
   hash2 = "faab883598c8c379acfd0b9dccc93d0c"
   hash3 = "7c3a2225792be3087d6e8c073cb8a58d"
   hash4 = "8b08a91726ff8e4218af1336b4bf1f1d"
   hash5 = "6f055150861d8d6e145e9aca65f92822"
   hash6 = "59fe83e0ae12e085e0fa301ecca6776f"
   sample_filetype = "exe"
   yaragenerator = "https://github.com/Xen0ph0n/YaraGenerator"
strings:
   $string0 = "setKey:"
   $string1 = "setSharedMemoryID:"
   $string2 = "writeMemorybyXPC:offset:fromComponent:"
   $string3 = "_getuid"
   $string4 = "/usr/lib/libSystem.B.dylib"
   $string5 = "/System/Library/Frameworks/Foundation.framework/Ver-
sions/C/Foundation"
   $string6 = "/tmp/launchch-%d"
   $string7 = "/usr/lib/system/libsystem_sandbox.dylib"
   $string8 = "readMemory:fromComponent:"
   $string9 = "mSemaphoreName"
   $string10 = "fileExistsAtPath:"
   $string11 = "initWithFormat:"
   $string12 = "__NSConcreteGlobalBlock"
   $string13 = "mAmIPrivUser"
   $string14 = "mSharedMemoryID"
   $string15 = "mSemaphoreID"
   $string16 = "com.apple."
condition:
   16 of them
}
```

5.2 Case study and evaluation of the plugin

The enhanced plugin was tested against multiple instances of a virtual machine running OS X with Bitcoin clients installed. Following is a showcase of some of the results.

The Hive Bitcoin client for desktop was one of the tests. A wallet was created and three Bitcoin addresses were used, two of whom are shown in Figure 5-1 that follows.

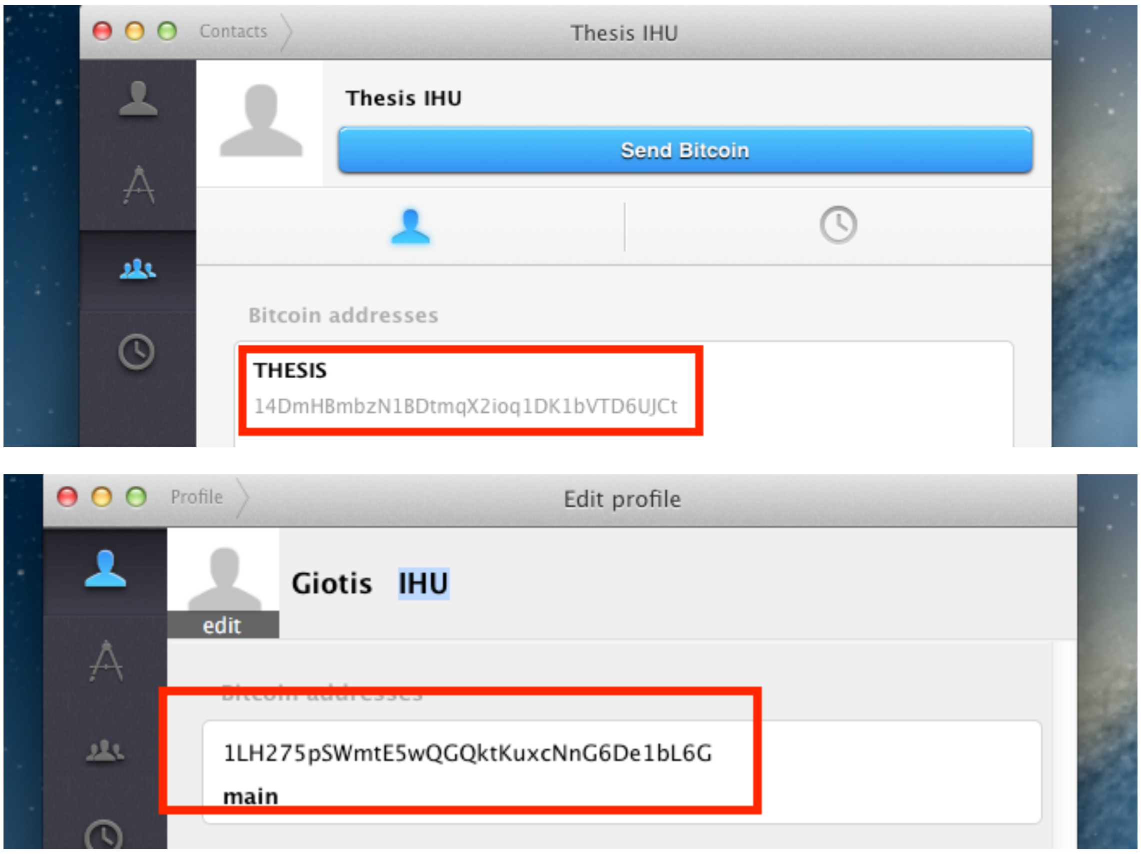
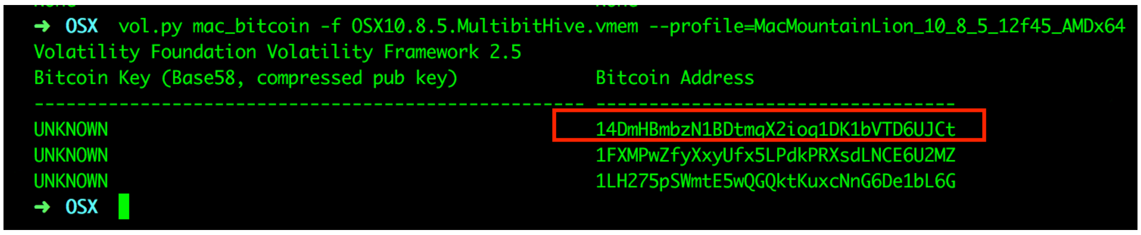


Figure 5-1, The Hive Bitcoin client

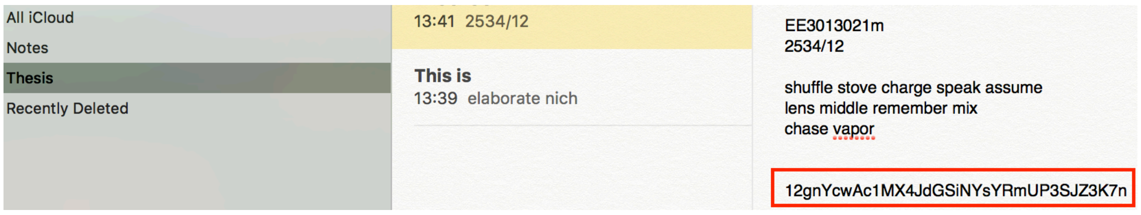
Running Volatility against the memory of the system resulted in all three Bitcoin addresses revealed to the investigator, as show in the following Figure



```
➜  OSX  vol.py mac_bitcoin -f OSX10.8.5.MultibitHive.vmem --profile=MacMountainLion_10_8_5_12f45_AMDx64
Volatility Foundation Volatility Framework 2.5
Bitcoin Key (Base58, compressed pub key)            Bitcoin Address
--------------------------------------------------- ----------------------------------
UNKNOWN                                             14DmHBmbzN1BDtmqX2ioq1DK1bVTD6UJCt
UNKNOWN                                             1FXMPwZfyXxyUfx5LPdkPRXsdLNCE6U2MZ
UNKNOWN                                             1LH275pSWmtE5wQGQktKuxcNnG6De1bL6G
➜  OSX
```

Figure 5-2, Bitcoin addresses of Hive revealed

Moreover, apart from Bitcoin clients an investigator may a also search for other processes by adding the name of the process to the *bccclients* list. For example, the native application of notes for OS X:


All iCloud
Notes
Thesis
Recently Deleted

13:41 2534/12

This is
13:39 elaborate nich

EE3013021m
2534/12

shuffle stove charge speak assume
lens middle remember mix
chase vapor

12gnYcwAc1MX4JdGSiNYsYRmUP3SJZ3K7n

Figure 5-3, A Bitcoin wallet address in *Notes*

and the detection of the Bitcoin key by Volatility:

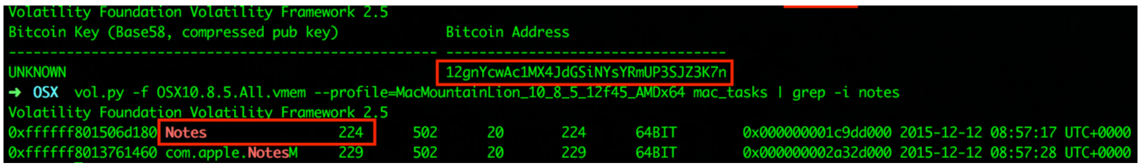

```
Volatility Foundation Volatility Framework 2.5
Bitcoin Key (Base58, compressed pub key)            Bitcoin Address
--------------------------------------------------- ----------------------------------
UNKNOWN                                             12gnYcwAc1MX4JdGSiNYsYRmUP3SJZ3K7n
➜ OSX  vol.py -f OSX10.8.5.All.vmem --profile=MacMountainLion_10_8_5_12f45_AMDx64 mac_tasks | grep -i notes
Volatility Foundation Volatility Framework 2.5
0xffffff801506d180 Notes             224     502     20      224     64BIT       0x000000001c9dd000 2015-12-12 08:57:17 UTC+0000
0xffffff8013761460 com.apple.NotesM  229     502     20      229     64BIT       0x000000002a32d000 2015-12-12 08:57:28 UTC+0000
```

Figure 5-4, Recovering an address from *Notes*

Yara signatures are implemented in a similar way. Following is the snippet of code for detected the DevilRobber malware in a process[115]:

```
            bit_addrs = []
                    devil_rule = yara.compile(sources = {'n' : 'rule r1 {meta: author = "Giotis"
date = "2015-11-10" description = "OSX" hash0 =
"a123cc209802aa37cc62d23682e8cf8d" hash1 =
"16e3bc0415056eb15b2752613970b79d" hash2 =
"9e22ceb19f397f8cd018f90c857ab638" sample_filetype = "exe" yaragenerator =
"https://github.com/Xen0ph0n/YaraGenerator strings: $string0 = "_read$UNIX2003"
$string1 = "content-length" $string2 = "URLBase" $string3 = "/usr/lib/dyld" $string4 =
"[FF02::C]" $string5 = "_listen$UNIX2003" $string6 = "_curl_global_init" $string7 =
"_close$UNIX2003" $string8 = "_malloc" $string9 = "__const" $string10 = "rm -f s.txt"
$string11 = "_inet_addr" $string12 = "presentationURL" $string13 = "__common"
$string14 = "_recv$UNIX2003" $string15 = "_curl_easy_init" $string16 = "__De-
faultRuneLocale" $string17 = "_freeaddrinfo" $string18 = "_strcmp" condition: 18 of
them}'})
                    for task in bit_tasks:
                        scanner = mac_yarascan.MapYaraScanner(task = task, rules =
devil_rule)
                        for hit, address in scanner.scan():
                            content = scanner.address_space.zread(address, 34)
                                bit_addrs.append('DevilRobber detected!')
```

[115] The Yara rule for DevilRobber was created earlier, at Chapter 5.1

5.3 Strong and weak points of the implementation

By expanding the list an investigator can virtually examine every process for artifacts associated with Bitcoin. One can even implement Python's regular expressions for detecting random names usually associated with malware. This however would require a great amount of CPU resources and can be time consuming, making this a non-optimal solution for investigations where the systems used to perform the analysis are not state of the art from their CPU perspective.

Yara rules for detecting malware are also an efficient way of dealing malware like Devil Robber that cannot be detected due to the random names it uses. Like in the previous case, they have the disadvantage of having to be updated along with the updates of the malware itself and their effectiveness is directly connected with the number of samples used for their generation: The merrier the better.

A implementation in the plugin of both ways of detection for each process of interest, along with an update mechanism is the next step towards a more efficient way of performing analysis focused on Bitcoin related processes.

6 Chapter 6 - Conclusions

Summary of the thesis and future development

6.1 Summary

For this thesis we examined the current state of malware and malware economy. We focused on a portion of malware programs, closely related to a recently introduced innovation, digital currencies. We examined the current state of digital currencies, focusing at the predominant of them, Bitcoin, and compared it with other digital currencies. We introduced the way it works, why is it considered by some the future of global economy. We also took a close look on how criminals are combining those two technologies, Bitcoin and malware, in order to locate and take advantage of both unsuspecting and educated victims.

We advocated on the need of digital forensics a section of forensics introduced officially in the early ‘80s, a lot later than the creation of computers and other forensic sciences. We debated on why memory forensics are a vital part of digital forensics and researched the inner workings of one of the cornerstone frameworks of memory forensics, the Volatility Framework.

We investigated the current state of plugins for the Volatility framework, and enhanced it by implementing a series of additions regarding Bitcoin related legitimate programs and malware, after inspecting a variety of them. Finally, we propose a series of improvements that will further expand and advance the field of memory forensics against malware.

6.2 Contribution

All the code created for this thesis along with future development, will be made available as an open-source project, in a public repository[116] under the MIT Licence[117]

6.3 Future Development

Creating equivalent plugins for Windows and Linux is considered the top priority regarding future development. Adding multithreading or multiprocessing futures to the plugin would greatly improve the experience of a forensic investigator who is working a home computer. Moreover, multiprocessing support will be the gateway to another improvement that as of now is not cost and time effective; The examination for Bitcoin related artifacts in every process that is found in a memory dump, by automatically loading its PID into a list similar to the one introduced in the thesis and then separately analyzing them. Furthermore, combining Yara rules in the above scenario will result in an all around solution for detecting Bitcoin related malware.

[116] https://github.com/GiotisD
[117] https://opensource.org/licenses/MIT

7 Bibliography

[1] N. Losses, "Estimating the Global Cost of Cybercrime," *McAfee, Cent. Strateg. Int. Stud.*, 2014.

[2] "Cyber crime costs global economy $445 bn annually." [Online]. Available: http://www.telegraph.co.uk/technology/internet-security/10886640/Cyber-crime-costs-global-economy-445-bn-annually.html. [Accessed: 16-Jun-2015].

[3] E. Mills, "Cybercrime Cost Firms $1 Trillion Globally," 2009.

[4] D. Huang, "Profit-driven abuses of virtual currencies," *Univ. California, San Diego*, 2013.

[5] H. Dharmdasani, "Botnets and Crypto Currency-Effects of Botnets on the Bitcoin Ecosystem," 2013.

[6] K. Poulsen, "New malware steals your bitcoin," 2011.

[7] A. Chiang, "Bitcoin-mining Malware is rising in APAC region," 2013. [Online]. Available: http://apac.trendmicro.com/apac/about-us/newsroom/releases/articles/20131224091333.html. [Accessed: 10-Sep-2015].

[8] M. Spagnuolo, F. Maggi, and S. Zanero, "Bitiodine: Extracting intelligence from the bitcoin network," *Financ. Cryptogr. Data ...*, 2014.

[9] L. Abrams, "CryptoLocker Ransomware Information Guide and FAQ," *Viitattu*, 2013.

[10] P. Outbreaks, "Malicious-Advertising Attacks Inflict Ransomware on Victims," *ieeexplore.ieee.org*.

[11] H. Orman, "The Morris worm: A fifteen-year perspective," *IEEE Secur. Priv.*, 2003.

[12] R. Morris, "The Morris Worm source code," 1988. [Online]. Available: http://www.foo.be/docs-free/morris-worm/worm/. [Accessed: 16-Oct-2015].

[13] Microsoft, "The Evolution of Malware and the Threat Landscape – a 10-Year review," 2012.

[14] C. Shannon and D. Moore, "The spread of the witty worm," *Secur. Privacy, IEEE*, 2004.

[15] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Secur. Privacy, IEEE*, 2011.

[16] N. Falliere, L. Murchu, and E. Chien, "W32. stuxnet dossier," *White Pap. Symantec Corp., Secur. ...*, 2011.

[17] J. M. Kizza, *Guide to Computer Network Security*. London: Springer London, 2015.

[18] M. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory*. 2014.

[19] M. Ligh, S. Adair, B. Hartstein, and M. Richard, *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. 2010.

[20] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 2012.

[21] A. Case, "Mac Memory Analysis with Volatility," *DFIR Summit*, 2012. [Online]. Available: https://reverse.put.as/wp-content/uploads/2011/06/sas-summit-mac-memory-analysis-with-volatility.pdf. [Accessed: 24-Aug-2015].

[22] M. KA, "Linux Memory Diff Analysis using Volatility," 2015. [Online]. Available: http://malware-unplugged.blogspot.in/2015/09/linux-memory-diff-analysis-using.html. [Accessed: 25-Sep-2015].

[23] A. Case, "Mac Memory Analysis with Volatility," 2011. [Online]. Available: https://digital-forensics.sans.org/summit-archives/2012/mac-memory-analysis-with-volatility.pdf. [Accessed: 24-Aug-2015].

[24] A. F. Hay, "Forensic Memory Analysis for Apple OS X." [Online]. Available: https://reverse.put.as/wp-content/uploads/2011/06/FORENSIC-MEMORY-ANALYSIS-FOR-APPLE-OS-X.pdf. [Accessed: 24-Aug-2015].

[25] J. Seitz, *Gray Hat Python: Python programming for hackers and reverse engineers*. 2009.

[26] Z. Shaw, "Learn Python the hard way," 2010.

[27] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Consulted*, 2008.

[28] J. DAVIS, "The Crypto-Currency," *The New Yorker*, 2011. [Online]. Available: http://www.newyorker.com/magazine/2011/10/10/the-crypto-currency. [Accessed: 11-Dec-2015].

[29] A. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. 2014.

[30] R. McMillan, "Ex-Googler Gives the World a Better Bitcoin | WIRED," *Wired*, 2013. [Online]. Available: http://www.wired.com/2013/08/litecoin/. [Accessed: 11-Dec-2015].

[31] A. Greenberg, "Darkcoin, the Shadowy Cousin of Bitcoin, Is Booming | WIRED," *Wired*, 2013. [Online]. Available: http://www.wired.com/2014/05/darkcoin-is-booming/. [Accessed: 11-Dec-2015].

[32] M. Swan, *Blockchain: Blueprint for a New Economy*. 2015.

[33] J. Bearman, "The Untold Story of Silk Road, Part 1 | WIRED," 2015. [Online]. Available: http://www.wired.com/2015/04/silk-road-1/. [Accessed: 23-Aug-2015].

[34] M. Moser, R. Bohme, and D. Breuker, "An inquiry into money laundering tools in the Bitcoin ecosystem," *eCrime Res. Summit (…*, 2013.

[35] A. Kujawa, "Potentially Unwanted Miners," *Malwarebytes Unpacked*, 2013.

[Online]. Available: https://blog.malwarebytes.org/fraud-scam/2013/11/potentially-unwanted-miners-toolbar-peddlers-use-your-system-to-make-btc/. [Accessed: 05-Dec-2015].

[36] J. Cannel, "Cryptolocker Ransomware: What You Need to Know," *Malwarebytes Unpacked*, 2013.

[37] G. O'Gorman and G. McDonald, *Ransomware: a growing menace*. 2012.

[38] J. Ami-Narh and P. Williams, "Digital forensics and the legal system: A dilemma of our times," *Aust. Digit. Forensics Conf.*, 2008.

[39] S. Vömel and J. Stüttgen, "An evaluation platform for forensic memory acquisition software," *Digit. Investig.*, 2013.

[40] "Welcome to Python.org." [Online]. Available: https://www.python.org/. [Accessed: 17-Oct-2015].

[41] Volatility Foundation, "An advanced memory forensics framework." .

[42] "ForensicsWiki." [Online]. Available: http://forensicswiki.org/wiki/Main_Page. [Accessed: 25-Aug-2015].

[43] A. Walters and N. Petroni, "Volatools: integrating volatile memory forensics into the digital investigation process. Blackhat Hat DC 2007," 2007.

[44] D. Olenick, "Apple iOS And Google Android Smartphone Market Share Flattening: IDC - Forbes," *Forbes.com LLC*, 2015. [Online]. Available: http://www.forbes.com/sites/dougolenick/2015/05/27/apple-ios-and-google-android-smartphone-market-share-flattening-idc/2/. [Accessed: 17-Oct-2015].

[45] Cem Gurkok, "The Volatility Foundation - Open Source Memory Forensics | 2014," 2014. [Online]. Available: http://www.volatilityfoundation.org/#!2014/cjpn. [Accessed: 25-Aug-2015].

[46] MoonSols, "MoonSols Windows Memory Toolkit | MoonSols." [Online]. Available: http://www.moonsols.com/windows-memory-toolkit/. [Accessed: 24-Aug-2015].

[47] J. Stuettgen, "OSXPmem - pmem - The OSX Pmem memory acquisition tool. - Pmem is a suite of memory acquisition tools. - Google Project Hosting." [Online]. Available: https://code.google.com/p/pmem/wiki/OSXPmem. [Accessed: 24-Aug-2015].

[48] R. Endsley, "Physical Memory Analysis with the LiME Linux Memory Extractor," 2012. [Online]. Available: https://www.linux.com/learn/tutorials/565969-physical-memory-analysis-with-the-lime-linux-memory-extractor. [Accessed: 24-Aug-2015].

[49] S. K. Paul Rubin, David MacKenzie, "dd(1): convert/copy file - Linux man page." 2010.

[50] Brendan Dolan-Gavitt, "pdbparse, a GPL-licensed library for parsing Microsoft PDB files." 2015.

[51] B. Dolan-Gavitt, "Brendan Dolan-Gavitt -- Home." [Online]. Available: http://www.cc.gatech.edu/~brendan/. [Accessed: 18-Oct-2015].

[52] VolatilityTeam, "VolatilityTeam - volatility - Volatility Development Team - An

advanced memory forensics framework," 2012. [Online]. Available: https://code.google.com/p/volatility/wiki/VolatilityTeam. [Accessed: 02-Nov-2015].

[53] C. Gurkok, "What's in your silicon?: Hooking IDT in OS X and Detection," 2013. [Online]. Available: http://siliconblade.blogspot.gr/2013/07/idt-hooks-and-detecting-them-in-osx.html. [Accessed: 03-Dec-2015].

[54] C. Gurkok, "What's in your silicon?: Back to Defense: Finding Hooks in OS X with Volatility," 2013. [Online]. Available: http://siliconblade.blogspot.gr/2013/07/back-to-defense-finding-hooks-in-os-x.html. [Accessed: 03-Dec-2015].

[55] C. Gurkok, "What's in your silicon?: Offensive Volatility: Messing with the OS X Syscall Table," 2013. [Online]. Available: http://siliconblade.blogspot.gr/2013/07/offensive-volatility-messing-with-os-x.html. [Accessed: 03-Dec-2015].

[56] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," *Proc. 2012 ACM Conf. …*, 2012.

[57] L. Foundation, "2014 Enterprise End User Report," 2014. [Online]. Available: https://www.linuxfoundation.org/publications/linux-foundation/linux-end-user-trends-report-2014. [Accessed: 04-Dec-2015].

[58] M. Butler, "Android: Changing the mobile landscape," *Pervasive Comput. IEEE*, 2011.

[59] M. Fontanini, "Average coder: Linux rootkit implementation," 2011. [Online]. Available: http://average-coder.blogspot.gr/2011/12/linux-rootkit.html. [Accessed: 04-Dec-2015].

[60] S. McCarty, "Architecting Containers Part 1: Why Understanding User Space vs. Kernel Space Matters | Red Hat Enterprise Linux Blog," *Red Hat Enterprize Linux Blog*, 2015. [Online]. Available: http://rhelblog.redhat.com/2015/07/29/architecting-containers-part-1-user-space-vs-kernel-space/. [Accessed: 04-Dec-2015].

[61] P. Mochel, "The sysfs filesystem," *Linux Symp.*, 2005.

[62] A. Case, "Analyzing Linux Kernel Rootkits with Volatility," *OMFW*, 2012. [Online]. Available: http://volatility-labs.blogspot.gr/2012/10/omfw-2012-analyzing-linux-kernel.html. [Accessed: 05-Dec-2015].

[63] P. Wardle, "Malware Persistence on OS X Yosemite | USA 2015 RSA Conference," 2015. [Online]. Available: https://www.rsaconference.com/events/us15/agenda/sessions/1591/malware-persistence-on-os-x-yosemite. [Accessed: 31-Aug-2015].

[64] P. Wardle, "Virus Bulletin : VB2014 - Methods of malware persistence on Mac OS X," 2014. [Online]. Available: https://www.virusbtn.com/conference/vb2014/abstracts/Wardle.xml. [Accessed: 31-Aug-2015].

[65] A. Case and G. Richard, "Advancing Mac OS X rootkit detection," *Digit. Investig.*, 2015.

[66] Virtualcurrency.com, “The current Bitcoin acceptance market - Payments Cards & Mobile,” 2014. [Online]. Available: http://www.paymentscardsandmobile.com/current-bitcoin-acceptance-market/. [Accessed: 19-Aug-2015].

[67] K. McMillan, “Gaming Company Fined $1M for Turning Customers Into Secret Bitcoin Army | WIRED,” *Wired*, 2013. [Online]. Available: http://www.wired.com/2013/11/e-sports/. [Accessed: 05-Dec-2015].

[68] P. Dimotikalis, “Bitcoin mining: The stupid way - Gi0’s Blog,” 2013. [Online]. Available: http://giot.is/bitcoin-mining-the-stupid-way/. [Accessed: 24-Aug-2015].

[69] P. Dimotikalis, “Lets Talk Bitcoin - Ponzis, Malware, and the Hashing Cartel,” *Let’s Talk Bitcoin*, 2013. [Online]. Available: https://letstalkbitcoin.com/e27-ponzis-malware-and-the-hashing-cartel/. [Accessed: 05-Dec-2015].

[70] B. Carrier and J. Grand, “A hardware-based memory acquisition procedure for digital investigations,” *Digit. Investig.*, 2004.

[71] X. Chen, J. Andersen, and Z. Mao, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” ... *Networks With ...*, 2008.

[72] K. Kendall and C. McMillan, “Practical malware analysis,” *Black Hat Conf. USA*, 2007.

[73] O. Bach, “Tinba: World’s Smallest Malware Has Big Bag of Nasty Tricks,” *Security Intelligence IBM*, 2015. [Online]. Available: https://securityintelligence.com/tinba-worlds-smallest-malware-has-big-bag-of-nasty-tricks/. [Accessed: 10-Dec-2015].

[74] J. Kirk, “The Darlloz Linux worm diversifies to mine cryptocurrencies | Computerworld,” *ComputerWorld*, 2014. [Online]. Available: http://www.computerworld.com/article/2488828/malware-vulnerabilities/the-darlloz-linux-worm-diversifies-to-mine-cryptocurrencies.html. [Accessed: 10-Dec-2015].

8 Appendix A

Source code

bitcoin.py

```
# Volatility
# Copyright (C) 2007-2013 Volatility Foundation
#
# This file is part of Volatility.
#
# Volatility is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License Version 2 as
# published by the Free Software Foundation.  You may not use, modify or
# distribute this program under any other version of the GNU General
# Public License.
#
# Volatility is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with Volatility.  If not, see <http://www.gnu.org/licenses/>.
#

"""
@author:       Cem Gurkok
@license:      GNU General Public License 2.0
@contact:      cemgurkok@gmail.com
@organization:
"""

import re
import volatility.obj as obj
import volatility.plugins.mac.common as common
import volatility.plugins.mac.pstasks as pstasks
import volatility.debug as debug
import volatility.utils as utils
import volatility.plugins.mac.mac_yarascan as mac_yarascan
```

```
try:
   import pycoin.key as pykey
   import pycoin.encoding as pyenc
except ImportError:
   print "You need to install pycoin for this plugin to run [pip install pycoin]"

try:
   import yara
except ImportError:
   print "You need to install yara for this plugin to run
[https://github.com/plusvic/yara]"


class mac_bitcoin(common.AbstractMacCommand):
   """Get bitcoin artifacts from OS X multibit client memory"""

   def  __init__(self, config, *args, **kwargs):
      common.AbstractMacCommand.__init__(self, config, *args, **kwargs)
      self._config.add_option('PID', short_option = 'p', default = None, help = 'Operate
on these Process IDs (comma-separated)', action = 'store', type = 'str')

   def calculate(self):
      all_tasks = pstasks.mac_tasks(self._config).allprocs()
      bit_tasks = []

      try:
         if self._config.PID:
            # find tasks given PIDs
            pidlist = [int(p) for p in self._config.PID.split(',')]
            bit_tasks = [t for t in all_tasks if t.p_pid in pidlist]
         else:
            # find multibit process
            name_re = re.compile("JavaApplicationS", re.I)
            bit_tasks = [t for t in all_tasks if name_re.search(str(t.p_comm))]
      except:
         pass

      if len(bit_tasks) == 0:
         yield (None, None)


      # scan for bitcoin addresses with yara, 34 chars, https://en.bitcoin.it/wiki/Address
      # Most Bitcoin addresses are 34 characters. They consist of random digits and up-
percase
      # and lowercase letters, with the exception that the uppercase letter "O", upper-
case
      # letter "I", lowercase letter "l", and the number "0" are never used to prevent vis-
ual ambiguity.
      bit_addrs = []
      addr_rule = yara.compile(sources = {'n' : 'rule r1  {strings: $a = /[1-9a-zA-
z]{34}(?!OIl)/ condition: $a}'})
        for task in bit_tasks:
            scanner = mac_yarascan.MapYaraScanner(task = task, rules = addr_rule)
            for hit, address in scanner.scan():
                content = scanner.address_space.zread(address, 34)
                if pyenc.is_valid_bitcoin_address(content) and content not in bit_addrs:
                    bit_addrs.append(content)

        # scan for bitcoin keys with yara, 52 char compressed base58, starts with L or K,
https://en.bitcoin.it/wiki/Private_key
        addr_key = {}
        key_rule = yara.compile(sources = {'n' : 'rule r1  {strings: $a = /(L|K)[0-9A-Za-
z]{51}/ condition: $a}'})
        for task in bit_tasks:
            scanner = mac_yarascan.MapYaraScanner(task = task, rules = key_rule)
            for hit, address in scanner.scan():
                content = scanner.address_space.zread(address, 52)
                if pyenc.is_valid_wif(content):
                    secret_exp = pyenc.wif_to_secret_exponent(content)
                    key = pykey.Key(secret_exponent = secret_exp,is_compressed=True)
                    if key.address() not in addr_key.keys():
                        addr_key[key.address()] = content
                        yield(content, key.address())

        # addresses with no known keys
        for bit_addr in bit_addrs:
            if bit_addr not in addr_key.keys():
                yield ("UNKNOWN", bit_addr)

    def render_text(self, outfd, data):
        self.table_header(outfd, [("Bitcoin Key (Base58, compressed pub key)",
"<52"),("Bitcoin Address","<34")])
        for key, address in data:
            self.table_row(outfd, key, address)
```

unlocker.py

```
The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EX-
PRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MER-
CHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN
THE SOFTWARE.

vSMC Header Structure
Offset  Length  struct Type Description
----------------------------------------
0x00/00 0x08/08 Q      ptr  Offset to key table
0x08/08 0x04/4  I      int  Number of private keys
0x0C/12 0x04/4  I      int  Number of public keys

vSMC Key Data Structure
Offset  Length  struct Type Description
----------------------------------------
0x00/00 0x04/04 4s     int  Key name (byte reversed e.g. #KEY is YEK#)
0x04/04 0x01/01 B      byte Length of returned data
0x05/05 0x04/04 4s     int  Data type (byte reversed e.g. ui32 is 23iu)
0x09/09 0x01/01 B      byte Flag R/W
0x0A/10 0x06/06 6x     byte Padding
0x10/16 0x08/08 Q      ptr  Internal VMware routine
0x18/24 0x30/48 48B    byte Data
"""

import os
import sys
import struct
import subprocess

if sys.version_info < (2, 7):
    sys.stderr.write('You need Python 2.7 or later\n')
    sys.exit(1)

# Setup imports depending on whether IronPython or CPython
if sys.platform == 'win32' \
        or sys.platform == 'cli':
    from _winreg import *
```

```
def rot13(s):
    chars = 'AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz'
    trans = chars[26:] + chars[:26]
    rot_char = lambda c: trans[chars.find(c)] if chars.find(c) > -1 else c
    return ''.join(rot_char(c) for c in s)


def bytetohex(bytestr):
    return ''.join(['%02X ' % ord(x) for x in bytestr]).strip()


def printkey(i, offset, smc_key, smc_data):
    print str(i + 1).zfill(3) \
          + ' ' + hex(offset) \
          + ' ' + smc_key[0][::-1] \
          + ' ' + str(smc_key[1]).zfill(2) \
          + ' ' + smc_key[2][::-1].replace('\x00', ' ') \
          + ' ' + '{0:#0{1}x}'.format(smc_key[3], 4) \
          + ' ' + hex(smc_key[4]) \
          + ' ' + bytetohex(smc_data)


E_CLASS64 = 2
E_SHT_RELA = 4


def patchELF(f, oldOffset, newOffset):
    f.seek(0)
    magic = f.read(4)
    if not magic == b'\x7fELF':
        raise Exception('Magic number does not match')

    ei_class = struct.unpack('=B', f.read(1))[0]
    if ei_class != E_CLASS64:
        raise Exception('Not 64bit elf header: ' + ei_class)

    f.seek(40)
    e_shoff = struct.unpack('=Q', f.read(8))[0]
    f.seek(58)
    e_shentsize = struct.unpack('=H', f.read(2))[0]
    e_shnum = struct.unpack('=H', f.read(2))[0]
    e_shstrndx = struct.unpack('=H', f.read(2))[0]

    # print 'e_shoff: 0x{:x} e_shentsize: 0x{:x} e_shnum:0x{:x} e_shstrndx:0x{:x}'.for-
mat(e_shoff, e_shentsize, e_shnum, e_shstrndx)

    for i in range(0, e_shnum):
        f.seek(e_shoff + i * e_shentsize)
        e_sh = struct.unpack('=LLQQQQLLQQ', f.read(e_shentsize))
        e_sh_name = e_sh[0]
        e_sh_type = e_sh[1]
        e_sh_offset = e_sh[4]
        e_sh_size = e_sh[5]
        e_sh_entsize = e_sh[9]
        if e_sh_type == E_SHT_RELA:
            e_sh_nument = e_sh_size / e_sh_entsize
            # print 'RELA at 0x{:x} with {:d} entries'.format(e_sh_offset, e_sh_nument)
            for j in range(0, e_sh_nument):
                f.seek(e_sh_offset + e_sh_entsize * j)
                rela = struct.unpack('=QQq', f.read(e_sh_entsize))
                r_offset = rela[0]
                r_info = rela[1]
                r_addend = rela[2]
                if r_addend == oldOffset:
                    r_addend = newOffset
                    f.seek(e_sh_offset + e_sh_entsize * j)
                    f.write(struct.pack('=QQq', r_offset, r_info, r_addend))
                    print 'Relocation modified at: ' + hex(e_sh_offset + e_sh_entsize * j)


def patchkeys(f, vmx, key, osname):
    # Setup struct pack string
    key_pack = '=4sB4sB6xQ'
    smc_old_memptr = 0
    smc_new_memptr = 0

    # Do Until OSK1 read
    i = 0
    while True:

        # Read key into struct str and data byte str
        offset = key + (i * 72)
        f.seek(offset)
        smc_key = struct.unpack(key_pack, f.read(24))
        smc_data = f.read(smc_key[1])

        # Reset pointer to beginning of key entry
        f.seek(offset)

        if smc_key[0] == 'SKL+':
            # Use the +LKS data routine for OSK0/1
            smc_new_memptr = smc_key[4]
            print '+LKS Key: '
            printkey(i, offset, smc_key, smc_data)

        elif smc_key[0] == '0KSO':
            # Write new data routine pointer from +LKS
            print 'OSK0 Key Before:'
            printkey(i, offset, smc_key, smc_data)
            smc_old_memptr = smc_key[4]
            f.seek(offset)
            f.write(struct.pack(key_pack, smc_key[0], smc_key[1], smc_key[2],
smc_key[3], smc_new_memptr))
            f.flush()

            # Write new data for key
            f.seek(offset + 24)
            smc_new_data = rot13('bheuneqjbexolgurfrjbeqfthneqrqcy')
            f.write(smc_new_data)
            f.flush()

            # Re-read and print key
            f.seek(offset)
            smc_key = struct.unpack(key_pack, f.read(24))
            smc_data = f.read(smc_key[1])
            print 'OSK0 Key After:'
            printkey(i, offset, smc_key, smc_data)

        elif smc_key[0] == '1KSO':
            # Write new data routine pointer from +LKS
            print 'OSK1 Key Before:'
            printkey(i, offset, smc_key, smc_data)
            smc_old_memptr = smc_key[4]
            f.seek(offset)
            f.write(struct.pack(key_pack, smc_key[0], smc_key[1], smc_key[2],
smc_key[3], smc_new_memptr))
            f.flush()

            # Write new data for key
            f.seek(offset + 24)
            smc_new_data = rot13('rnfrqbagfgrny(p)NccyrPbzchgreVap')
            f.write(smc_new_data)
            f.flush()

            # Re-read and print key
            f.seek(offset)
            smc_key = struct.unpack(key_pack, f.read(24))
            smc_data = f.read(smc_key[1])
            print 'OSK1 Key After:'
            printkey(i, offset, smc_key, smc_data)

            # Finished so get out of loop
            break

        else:
            pass

        i += 1
```

```
    return smc_old_memptr, smc_new_memptr


def patchsmc(name, osname, sharedobj):
    with open(name, 'r+b') as f:

        smc_old_memptr = 0
        smc_new_memptr = 0

        # Read file into string variable
        vmx = f.read()

        print 'File: ' + name

        # Setup hex string for vSMC headers
        # These are the private and public key counts
        smc_header_v0 = '\xF2\x00\x00\x00\xF0\x00\x00\x00'
        smc_header_v1 = '\xB4\x01\x00\x00\xB0\x01\x00\x00'

        # Setup hex string for #KEY key
        key_key = '\x59\x45\x4B\x23\x04\x32\x33\x69\x75'

        # Setup hex string for $Adr key
        adr_key = '\x72\x64\x41\x24\x04\x32\x33\x69\x75'

        # Find the vSMC headers
        smc_header_v0_offset = vmx.find(smc_header_v0) - 8
        smc_header_v1_offset = vmx.find(smc_header_v1) - 8

        # Find '#KEY' keys
        smc_key0 = vmx.find(key_key)
        smc_key1 = vmx.rfind(key_key)

        # Find '$Adr' key only V1 table
        smc_adr = vmx.find(adr_key)

        # Print vSMC0 tables and keys
        print 'appleSMCTableV0 (smc.version = "0")'
        print 'appleSMCTableV0 Address      : ' + hex(smc_header_v0_offset)
        print 'appleSMCTableV0 Private Key #: 0xF2/242'
        print 'appleSMCTableV0 Public Key  #: 0xF0/240'

        if (smc_adr - smc_key0) != 72:
            print 'appleSMCTableV0 Table        : ' + hex(smc_key0)
            smc_old_memptr, smc_new_memptr = patchkeys(f, vmx, smc_key0, osname)
        elif (smc_adr - smc_key1) != 72:
            print 'appleSMCTableV0 Table        : ' + hex(smc_key1)
            smc_old_memptr, smc_new_memptr = patchkeys(f, vmx, smc_key1, osname)

        print
        # Print vSMC1 tables and keys
        print 'appleSMCTableV1 (smc.version = "1")'
        print 'appleSMCTableV1 Address      : ' + hex(smc_header_v1_offset)
        print 'appleSMCTableV1 Private Key #: 0x01B4/436'
        print 'appleSMCTableV1 Public Key  #: 0x01B0/432'

        if (smc_adr - smc_key0) == 72:
            print 'appleSMCTableV1 Table        : ' + hex(smc_key0)
            smc_old_memptr, smc_new_memptr = patchkeys(f, vmx, smc_key0, osname)
        elif (smc_adr - smc_key1) == 72:
            print 'appleSMCTableV1 Table        : ' + hex(smc_key1)
            smc_old_memptr, smc_new_memptr = patchkeys(f, vmx, smc_key1, osname)

        print

        # Find matching RELA record in .rela.dyn in ESXi ELF files
        # This is temporary code until proper ELF parsing written
        if sharedobj:
            print 'Modifying RELA records from: ' + hex(smc_old_memptr) + ' to ' +
hex(smc_new_memptr)
            patchELF(f, smc_old_memptr, smc_new_memptr)

        # Tidy up
        f.flush()
        f.close()


def patchbase(name):
    # Patch file
    print 'GOS Patching: ' + name
    f = open(name, 'r+b')

    # Entry to search for in GOS table
    darwin = (
        '\x10\x00\x00\x00\x10\x00\x00\x00'
        '\x02\x00\x00\x00\x00\x00\x00\x00'
        '\x00\x00\x00\x00\x00\x00\x00\x00'
        '\x00\x00\x00\x00\x00\x00\x00\x00'
        '\xBE'
    )

    # Read file into string variable
    base = f.read()

    # Loop thorugh each entry and set top bit
    # 0xBE --> 0xBF
    offset = 0
    while offset < len(base):
        offset = base.find(darwin, offset)
        if offset == -1:
            break
        f.seek(offset + 32)
        flag = f.read(1)
        if flag == '\xBE':
            f.seek(offset + 32)
            f.write('\xBF')
            print 'GOS Patched flag @: ' + hex(offset)
        else:
            print 'GOS Unknown flag @: ' + hex(offset) + '/' + hex(int(flag))

        offset += 33

    # Tidy up
    f.flush()
    f.close()
    print 'GOS Patched: ' + name


def patchvmkctl(name):
    # Patch file
    print 'smcPresent Patching: ' + name
    f = open(name, 'r+b')

    # Read file into string variable
    vmkctl = f.read()
    applesmc = vmkctl.find('applesmc')
    f.seek(applesmc)
    f.write('vmkernel')

    # Tidy up
    f.flush()
    f.close()
    print 'smcPresent Patched: ' + name


def main():
    # Work around absent Platform module on VMkernel
    if os.name == 'nt' or os.name == 'cli':
        osname = 'windows'
    else:
        osname = os.uname()[0].lower()

    vmx_so = False

    # Setup default paths
    if osname == 'darwin':
        vmx_path = '/Applications/VMware Fusion.app/Contents/Library/'
        vmx = vmx_path + 'vmware-vmx'
        vmx_debug = vmx_path + 'vmware-vmx-debug'
        vmx_stats = vmx_path + 'vmware-vmx-stats'
        vmwarebase = ''
        libvmkctl = ''

    elif osname == 'linux':
        vmx_path = '/usr/lib/vmware/bin/'
        vmx = vmx_path + 'vmware-vmx'
        vmx_debug = vmx_path + 'vmware-vmx-debug'
        vmx_stats = vmx_path + 'vmware-vmx-stats'
        vmx_version = subprocess.check_output(["vmplayer", "-v"])
        if vmx_version.startswith('VMware Player 12'):
            vmx_so = True
            vmwarebase = '/usr/lib/vmware/lib/libvmwarebase.so/libvmwarebase.so'
        else:
            vmwarebase = '/usr/lib/vmware/lib/libvmwarebase.so.0/libvmwarebase.so.0'
        libvmkctl = ''

    elif osname == 'vmkernel':
        vmx_path = '/unlocker/'
        vmx = vmx_path + 'vmx'
        vmx_debug = vmx_path + 'vmx-debug'
        vmx_stats = vmx_path + 'vmx-stats'
        vmx_so = True
        vmwarebase = ''
        libvmkctl = vmx_path + 'libvmkctl.so'

    elif osname == 'windows':
        reg = ConnectRegistry(None, HKEY_LOCAL_MACHINE)
        key = OpenKey(reg, r'SOFTWARE\Wow6432Node\VMware, Inc.\VMware
Workstation')
        vmwarebase_path = QueryValueEx(key, 'InstallPath')[0]
        vmx_path = QueryValueEx(key, 'InstallPath64')[0]
        vmx = vmx_path + 'vmware-vmx.exe'
        vmx_debug = vmx_path + 'vmware-vmx-debug.exe'
        vmx_stats = vmx_path + 'vmware-vmx-stats.exe'
        vmwarebase = vmwarebase_path + 'vmwarebase.dll'
        libvmkctl = ''

    else:
        print('Unknown Operating System: ' + osname)
        return

    # Patch the vmx executables skipping stats version for Player
    patchsmc(vmx, osname, vmx_so)
    patchsmc(vmx_debug, osname, vmx_so)
    try:
        patchsmc(vmx_stats, osname, vmx_so)
    except IOError:
        pass
    # Patch vmwarebase for Workstation and Player
    # Not required on Fusion or ESXi as table already has correct flags
    if vmwarebase != '':
        patchbase(vmwarebase)
    else:
        print 'Patching vmwarebase is not required on this system'

    if osname == 'vmkernel':
        patchvmkctl(libvmkctl)


if __name__ == '__main__':
    main()
```

List of plugins included in Volatility Framework 2.5

amcache	- Print AmCache information
apihooks	- Detect API hooks in process and kernel memory
atoms	- Print session and window station atom tables
atomscan	- Pool scanner for atom tables
auditpol	- Prints out the Audit Policies from HKLM\SECURITY\Policy\PolAdtEv
bigpools	- Dump the big page pools using BigPagePoolScanner
bioskbd	- Reads the keyboard buffer from Real Mode memory
cachedump	- Dumps cached domain hashes from memory
callbacks	- Print system-wide notification routines
clipboard	- Extract the contents of the windows clipboard
cmdline	- Display process command-line arguments
cmdscan	- Extract command history by scanning for _COMMAND_HISTORY
connections	- Print list of open connections [Windows XP and 2003 Only]
connscan	- Pool scanner for tcp connections

consoles	- Extract command history by scanning for _CONSOLE_INFORMATION
crashinfo	- Dump crash-dump information
deskscan	- Poolscaner for tagDESKTOP (desktops)
devicetree	- Show device tree
dlldump	- Dump DLLs from a process address space
dlllist	- Print list of loaded dlls for each process
driverirp	- Driver IRP hook detection
drivermodule	- Associate driver objects to kernel modules
driverscan	- Pool scanner for driver objects
dumpcerts	- Dump RSA private and public SSL keys
dumpfiles	- Extract memory mapped and cached files
dumpregistry	- Dumps registry files out to disk
editbox	- Dumps various data from ComCtl Edit controls (experimental: ListBox, ComboBox)
envars	- Display process environment variables
eventhooks	- Print details on windows event hooks
evtlogs	- Extract Windows Event Logs (XP/2003 only)
filescan	- Pool scanner for file objects
gahti	- Dump the USER handle type information
gditimers	- Print installed GDI timers and callbacks
gdt	- Display Global Descriptor Table
getservicesids	- Get the names of services in the Registry and return Calculated SID
getsids	- Print the SIDs owning each process
handles	- Print list of open handles for each process
hashdump	- Dumps passwords hashes (LM/NTLM) from memory

hibinfo	- Dump hibernation file information
hivedump	- Prints out a hive
hivelist	- Print list of registry hives.
hivescan	- Pool scanner for registry hives
hpakextract	- Extract physical memory from an HPAK file
hpakinfo	- Info on an HPAK file
idt	- Display Interrupt Descriptor Table
iehistory	- Reconstruct Internet Explorer cache / history
imagecopy	- Copies a physical address space out as a raw DD image
imageinfo	- Identify information for the image
impscan	- Scan for calls to imported functions
joblinks	- Print process job link information
kdbgscan	- Search for and dump potential KDBG values
kpcrscan	- Search for and dump potential KPCR values
ldrmodules	- Detect unlinked DLLs
limeinfo	- Dump Lime file format information
linux_apihooks	- Checks for userland apihooks
linux_arp	- Print the ARP table
linux_banner	- Prints the Linux banner information
linux_bash	- Recover bash history from bash process memory
linux_bash_env	- Recover a process' dynamic environment variables
linux_bash_hash	- Recover bash hash table from bash process memory
linux_check_afinfo	- Verifies the operation function pointers of network protocols
linux_check_creds	- Checks if any processes are sharing credential structures

linux_check_evt_arm	- Checks the Exception Vector Table to look for syscall table hooking
linux_check_fop	- Check file operation structures for rootkit modifications
linux_check_idt	- Checks if the IDT has been altered
linux_check_inline_kernel	- Check for inline kernel hooks
linux_check_modules	- Compares module list to sysfs info, if available
linux_check_syscall	- Checks if the system call table has been altered
linux_check_syscall_arm	- Checks if the system call table has been altered
linux_check_tty	- Checks tty devices for hooks
linux_cpuinfo	- Prints info about each active processor
linux_dentry_cache	- Gather files from the dentry cache
linux_dmesg	- Gather dmesg buffer
linux_dump_map	- Writes selected memory mappings to disk
linux_dynamic_env	- Recover a process' dynamic environment variables
linux_elfs	- Find ELF binaries in process mappings
linux_enumerate_files	- Lists files referenced by the filesystem cache
linux_find_file	- Lists and recovers files from memory
linux_getcwd	- Lists current working directory of each process
linux_hidden_modules	- Carves memory to find hidden kernel modules
linux_ifconfig	- Gathers active interfaces
linux_info_regs	- It's like 'info registers' in GDB. It prints out all the
linux_iomem	- Provides output similar to /proc/iomem
linux_kernel_opened_files	- Lists files that are opened from within the kernel
linux_keyboard_notifiers	- Parses the keyboard notifier call chain
linux_ldrmodules	- Compares the output of proc maps with the list of libraries from libdl

linux_library_list	- Lists libraries loaded into a process
linux_librarydump	- Dumps shared libraries in process memory to disk
linux_list_raw	- List applications with promiscuous sockets
linux_lsmod	- Gather loaded kernel modules
linux_lsof	- Lists file descriptors and their path
linux_malfind	- Looks for suspicious process mappings
linux_memmap	- Dumps the memory map for linux tasks
linux_moddump	- Extract loaded kernel modules
linux_mount	- Gather mounted fs/devices
linux_mount_cache	- Gather mounted fs/devices from kmem_cache
linux_netfilter	- Lists Netfilter hooks
linux_netscan	- Carves for network connection structures
linux_netstat	- Lists open sockets
linux_pidhashtable	- Enumerates processes through the PID hash table
linux_pkt_queues	- Writes per-process packet queues out to disk
linux_plthook	- Scan ELF binaries' PLT for hooks to non-NEEDED images
linux_proc_maps	- Gathers process memory maps
linux_proc_maps_rb	- Gathers process maps for linux through the mappings red-black tree
linux_procdump	- Dumps a process's executable image to disk
linux_process_hollow	- Checks for signs of process hollowing
linux_psaux	- Gathers processes along with full command line and start time
linux_psenv	- Gathers processes along with their static environment variables
linux_pslist	- Gather active tasks by walking the task_struct->task list

linux_pslist_cache	- Gather tasks from the kmem_cache
linux_pstree	- Shows the parent/child relationship between processes
linux_psxview	- Find hidden processes with various process listings
linux_recover_filesystem	- Recovers the entire cached file system from memory
linux_route_cache	- Recovers the routing cache from memory
linux_sk_buff_cache	- Recovers packets from the sk_buff kmem_cache
linux_slabinfo	- Mimics /proc/slabinfo on a running machine
linux_strings	- Match physical offsets to virtual addresses (may take a while, VERY verbose)
linux_threads	- Prints threads of processes
linux_tmpfs	- Recovers tmpfs filesystems from memory
linux_truecrypt_passphrase	- Recovers cached Truecrypt passphrases
linux_vma_cache	- Gather VMAs from the vm_area_struct cache
linux_volshell	- Shell in the memory image
linux_yarascan	- A shell in the Linux memory image
lsadump	- Dump (decrypted) LSA secrets from the registry
mac_adium	- Lists Adium messages
mac_apihooks	- Checks for API hooks in processes
mac_apihooks_kernel	- Checks to see if system call and kernel functions are hooked
mac_arp	- Prints the arp table
mac_bash	- Recover bash history from bash process memory
mac_bash_env	- Recover bash's environment variables
mac_bash_hash	- Recover bash hash table from bash process memory
mac_calendar	- Gets calendar events from Calendar.app
mac_check_mig_table	- Lists entires in the kernel's MIG table

mac_check_syscall_shadow	- Looks for shadow system call tables
mac_check_syscalls	- Checks to see if system call table entries are hooked
mac_check_sysctl	- Checks for unknown sysctl handlers
mac_check_trap_table	- Checks to see if mach trap table entries are hooked
mac_compressed_swap	- Prints Mac OS X VM compressor stats and dumps all compressed pages
mac_contacts	- Gets contact names from Contacts.app
mac_dead_procs	- Prints terminated/de-allocated processes
mac_dead_sockets	- Prints terminated/de-allocated network sockets
mac_dead_vnodes	- Lists freed vnode structures
mac_dmesg	- Prints the kernel debug buffer
mac_dump_file	- Dumps a specified file
mac_dump_maps	- Dumps memory ranges of process(es), optionally including pages in compressed swap
mac_dyld_maps	- Gets memory maps of processes from dyld data structures
mac_find_aslr_shift	- Find the ASLR shift value for 10.8+ images
mac_get_profile	- Automatically detect Mac profiles
mac_ifconfig	- Lists network interface information for all devices
mac_ip_filters	- Reports any hooked IP filters
mac_keychaindump	- Recovers possbile keychain keys. Use chainbreaker to open related keychain files
mac_ldrmodules	- Compares the output of proc maps with the list of libraries from libdl
mac_librarydump	- Dumps the executable of a process
mac_list_files	- Lists files in the file cache
mac_list_kauth_listeners	- Lists Kauth Scope listeners
mac_list_kauth_scopes	- Lists Kauth Scopes and their status

mac_list_raw	- List applications with promiscuous sockets
mac_list_sessions	- Enumerates sessions
mac_list_zones	- Prints active zones
mac_lsmod	- Lists loaded kernel modules
mac_lsmod_iokit	- Lists loaded kernel modules through IOkit
mac_lsmod_kext_map	- Lists loaded kernel modules
mac_lsof	- Lists per-process opened files
mac_machine_info	- Prints machine information about the sample
mac_malfind	- Looks for suspicious process mappings
mac_memdump	- Dump addressable memory pages to a file
mac_moddump	- Writes the specified kernel extension to disk
mac_mount	- Prints mounted device information
mac_netstat	- Lists active per-process network connections
mac_network_conns	- Lists network connections from kernel network structures
mac_notesapp	- Finds contents of Notes messages
mac_notifiers	- Detects rootkits that add hooks into I/O Kit (e.g. LogKext)
mac_orphan_threads	- Lists threads that don't map back to known modules/processes
mac_pgrp_hash_table	- Walks the process group hash table
mac_pid_hash_table	- Walks the pid hash table
mac_print_boot_cmdline	- Prints kernel boot arguments
mac_proc_maps	- Gets memory maps of processes
mac_procdump	- Dumps the executable of a process
mac_psaux	- Prints processes with arguments in user land (**argv)

mac_psenv (**envp)	- Prints processes with environment in user land
mac_pslist	- List Running Processes
mac_pstree	- Show parent/child relationship of processes
mac_psxview	- Find hidden processes with various process listings
mac_recover_filesystem	- Recover the cached filesystem
mac_route	- Prints the routing table
mac_socket_filters	- Reports socket filters
mac_strings	- Match physical offsets to virtual addresses (may take a while, VERY verbose)
mac_tasks	- List Active Tasks
mac_threads	- List Process Threads
mac_threads_simple	- Lists threads along with their start time and priority
mac_trustedbsd	- Lists malicious trustedbsd policies
mac_version	- Prints the Mac version
mac_volshell	- Shell in the memory image
mac_yarascan	- Scan memory for yara signatures
machoinfo	- Dump Mach-O file format information
malfind	- Find hidden and injected code
mbrparser	- Scans for and parses potential Master Boot Records
memdump	- Dump the addressable memory for a process
memmap	- Print the memory map
messagehooks	- List desktop and thread window message hooks
mftparser	- Scans for and parses potential MFT entries
moddump	- Dump a kernel driver to an executable file sample
modscan	- Pool scanner for kernel modules

modules	- Print list of loaded modules
multiscan	- Scan for various objects at once
mutantscan	- Pool scanner for mutex objects
netscan	- Scan a Vista (or later) image for connections and sockets
notepad	- List currently displayed notepad text
objtypescan	- Scan for Windows object type objects
patcher	- Patches memory based on page scans
poolpeek	- Configurable pool scanner plugin
pooltracker	- Show a summary of pool tag usage
printkey	- Print a registry key, and its subkeys and values
privs	- Display process privileges
procdump	- Dump a process to an executable file sample
pslist	- Print all running processes by following the EPROCESS lists
psscan	- Pool scanner for process objects
pstree	- Print process list as a tree
psxview	- Find hidden processes with various process listings
qemuinfo	- Dump Qemu information
raw2dmp dump	- Converts a physical memory sample to a windbg crash
screenshot	- Save a pseudo-screenshot based on GDI windows
servicediff	- List Windows services (ala Plugx)
sessions	- List details on _MM_SESSION_SPACE
shellbags	- Prints ShellBags info
shimcache	- Parses the Application Compatibility Shim Cache registry key
shutdowntime	- Print ShutdownTime of machine from registry

sockets	- Print list of open sockets
sockscan	- Pool scanner for tcp socket objects
ssdt	- Display SSDT entries
strings	- Match physical offsets to virtual addresses
svcscan	- Scan for Windows services
symlinkscan	- Pool scanner for symlink objects
thrdscan	- Pool scanner for thread objects
threads	- Investigate _ETHREAD and _KTHREADs
timeliner	- Creates a timeline from various artifacts in memory
timers	- Print kernel timers and associated module DPCs
truecryptmaster	- Recover TrueCrypt 7.1a Master Keys
truecryptpassphrase	- TrueCrypt Cached Passphrase Finder
truecryptsummary	- TrueCrypt Summary
unloadedmodules	- Print list of unloaded modules
userassist	- Print userassist registry keys and information
userhandles	- Dump the USER handle tables
vaddump	- Dumps out the vad sections to a file
vadinfo	- Dump the VAD info
vadtree	- Walk the VAD tree and display in tree format
vadwalk	- Walk the VAD tree
vboxinfo	- Dump virtualbox information
verinfo	- Prints out the version information from PE images
vmwareinfo	- Dump VMware VMSS/VMSN information
volshell	- Shell in the memory image
win10cookie	- Find the ObHeaderCookie value for Windows 10

windows	- Print Desktop Windows (verbose details)
wintree	- Print Z-Order Desktop Windows Tree
wndscan	- Pool scanner for window stations

List of out-of-the-box supported profiles in Volatility Framework 2.5

VistaSP0x64	- A Profile for Windows Vista SP0 x64
VistaSP0x86	- A Profile for Windows Vista SP0 x86
VistaSP1x64	- A Profile for Windows Vista SP1 x64
VistaSP1x86	- A Profile for Windows Vista SP1 x86
VistaSP2x64	- A Profile for Windows Vista SP2 x64
VistaSP2x86	- A Profile for Windows Vista SP2 x86
Win10x64	- A Profile for Windows 10 x64
Win10x86	- A Profile for Windows 10 x86
Win2003SP0x86	- A Profile for Windows 2003 SP0 x86
Win2003SP1x64	- A Profile for Windows 2003 SP1 x64
Win2003SP1x86	- A Profile for Windows 2003 SP1 x86
Win2003SP2x64	- A Profile for Windows 2003 SP2 x64
Win2003SP2x86	- A Profile for Windows 2003 SP2 x86
Win2008R2SP0x64	- A Profile for Windows 2008 R2 SP0 x64
Win2008R2SP1x64	- A Profile for Windows 2008 R2 SP1 x64
Win2008SP1x64	- A Profile for Windows 2008 SP1 x64
Win2008SP1x86	- A Profile for Windows 2008 SP1 x86
Win2008SP2x64	- A Profile for Windows 2008 SP2 x64
Win2008SP2x86	- A Profile for Windows 2008 SP2 x86

Win2012R2x64	- A Profile for Windows Server 2012 R2 x64
Win2012x64	- A Profile for Windows Server 2012 x64
Win7SP0x64	- A Profile for Windows 7 SP0 x64
Win7SP0x86	- A Profile for Windows 7 SP0 x86
Win7SP1x64	- A Profile for Windows 7 SP1 x64
Win7SP1x86	- A Profile for Windows 7 SP1 x86
Win81U1x64	- A Profile for Windows 8.1 Update 1 x64
Win81U1x86	- A Profile for Windows 8.1 Update 1 x86
Win8SP0x64	- A Profile for Windows 8 x64
Win8SP0x86	- A Profile for Windows 8 x86
Win8SP1x64	- A Profile for Windows 8.1 x64
Win8SP1x86	- A Profile for Windows 8.1 x86
WinXPSP1x64	- A Profile for Windows XP SP1 x64
WinXPSP2x64	- A Profile for Windows XP SP2 x64
WinXPSP2x86	- A Profile for Windows XP SP2 x86
WinXPSP3x86	- A Profile for Windows XP SP3 x86

List of profiles for OS X and Linux maintained by the Volatility Foundation

./Linux

```
 4    136 Aug 14  2014 CentOS

 4    136 Aug 14  2014 Debian

 4    136 Aug 14  2014 Fedora

 4    136 Aug 14  2014 OpenSUSE

 4    136 Aug 14  2014 RedHat
```

```
  4     136 Aug 14  2014 Ubuntu
```

./Linux/CentOS

```
  19     646 Aug 14  2014 x64
  19     646 Aug 14  2014 x86
```

./Linux/CentOS/x64

```
 1     363078 Aug 14  2014 CentOS50.zip
 1     368281 Aug 14  2014 CentOS51.zip
 1     405172 Aug 14  2014 CentOS510.zip
 1     371250 Aug 14  2014 CentOS52.zip
 1     377467 Aug 14  2014 CentOS53.zip
 1     389625 Aug 14  2014 CentOS54.zip
 1     393649 Aug 14  2014 CentOS55.zip
 1     398150 Aug 14  2014 CentOS56.zip
 1     400134 Aug 14  2014 CentOS57.zip
 1     402681 Aug 14  2014 CentOS58.zip
 1     404657 Aug 14  2014 CentOS59.zip
 1     614767 Aug 14  2014 CentOS60.zip
 1     628252 Aug 14  2014 CentOS61.zip
 1     634384 Aug 14  2014 CentOS62.zip
 1     637679 Aug 14  2014 CentOS63.zip
 1     653417 Aug 14  2014 CentOS64.zip
 1     673408 Aug 14  2014 CentOS65.zip
```

./Linux/CentOS/x86

```
 1     356003 Aug 14  2014 CentOS50.zip
 1     360890 Aug 14  2014 CentOS51.zip
```

```
1     391048 Aug 14  2014 CentOS510.zip

1     363549 Aug 14  2014 CentOS52.zip

1     369260 Aug 14  2014 CentOS53.zip

1     377998 Aug 14  2014 CentOS54.zip

1     381773 Aug 14  2014 CentOS55.zip

1     384813 Aug 14  2014 CentOS56.zip

1     386102 Aug 14  2014 CentOS57.zip

1     388581 Aug 14  2014 CentOS58.zip

1     390462 Aug 14  2014 CentOS59.zip

1     610104 Aug 14  2014 CentOS60.zip

1     623252 Aug 14  2014 CentOS61.zip

1     629071 Aug 14  2014 CentOS62.zip

1     637209 Aug 14  2014 CentOS63.zip

1     630904 Aug 14  2014 CentOS64.zip

1     649169 Aug 14  2014 CentOS65.zip
```

./Linux/Debian

```
 8     272 Sep 18 23:04 x64

 7     238 Aug 14  2014 x86
```

./Linux/Debian/x64

```
1     326904 Aug 14  2014 Debian40r9.zip

1     345487 Aug 14  2014 Debian5010.zip

1     503750 Aug 14  2014 Debian608.zip

1     626681 Aug 14  2014 Debian73.zip

1     626681 Aug 14  2014 Debian74.zip

1     750643 Sep 18 23:04 Debian8.zip
```

./Linux/Debian/x86

```
 1     305559 Aug 14  2014 Debian40r9.zip
 1     358396 Aug 14  2014 Debian5010.zip
 1     478922 Aug 14  2014 Debian608.zip
 1     603489 Aug 14  2014 Debian73.zip
 1     603394 Aug 14  2014 Debian74.zip
```

./Linux/Fedora

```
  18     612 Sep 18 23:04 x64
  17     578 Aug 14  2014 x86
```

./Linux/Fedora/x64

```
 1     438498 Aug 14  2014 Fedora10.zip
 1     495947 Aug 14  2014 Fedora11.zip
 1     560676 Aug 14  2014 Fedora12.zip
 1     591772 Aug 14  2014 Fedora13.zip
 1     631149 Aug 14  2014 Fedora14.zip
 1     676212 Aug 14  2014 Fedora15.zip
 1     703578 Aug 14  2014 Fedora16.zip
 1     687968 Aug 14  2014 Fedora17.zip
 1     709434 Aug 14  2014 Fedora18.zip
 1     726188 Aug 14  2014 Fedora19.zip
 1     744548 Aug 14  2014 Fedora20.zip
 1     836452 Sep 18 23:04 Fedora21Workstation.zip
 1     357541 Aug 14  2014 Fedora7.zip
 1     343550 Aug 14  2014 Fedora8.zip
 1     322097 Aug 14  2014 Fedora9.zip
```

```
1     361109 Aug 14  2014 FedoraCore6.zip
```

./Linux/Fedora/x86

```
1     425125 Aug 14  2014 Fedora10.zip
1     513673 Aug 14  2014 Fedora11.zip
1     550376 Aug 14  2014 Fedora12.zip
1     579463 Aug 14  2014 Fedora13.zip
1     609058 Aug 14  2014 Fedora14.zip
1     658152 Aug 14  2014 Fedora15.zip
1     686090 Aug 14  2014 Fedora16.zip
1     666869 Aug 14  2014 Fedora17.zip
1     719916 Aug 14  2014 Fedora18.zip
1     724117 Aug 14  2014 Fedora19.zip
1     724596 Aug 14  2014 Fedora20.zip
1     366486 Aug 14  2014 Fedora7.zip
1     352493 Aug 14  2014 Fedora8.zip
1     360395 Aug 14  2014 Fedora9.zip
1     352060 Aug 14  2014 FedoraCore6.zip
```

./Linux/OpenSUSE

```
 13     442 Aug 14  2014 x64
 13     442 Aug 14  2014 x86
```

./Linux/OpenSUSE/x64

```
1     324375 Aug 14  2014 OpenSUSE102.zip
1     335823 Aug 14  2014 OpenSUSE103.zip
1     369578 Aug 14  2014 OpenSUSE110.zip
1     433928 Aug 14  2014 OpenSUSE111.zip
```

```
1     549514 Aug 14  2014 OpenSUSE112.zip
1     568822 Aug 14  2014 OpenSUSE113.zip
1     629027 Aug 14  2014 OpenSUSE114.zip
1     708872 Aug 14  2014 OpenSUSE121.zip
1     694273 Aug 14  2014 OpenSUSE122.zip
1     715923 Aug 14  2014 OpenSUSE123.zip
1     746496 Aug 14  2014 OpenSUSE131.zip
```

./Linux/OpenSUSE/x86

```
1     319250 Aug 14  2014 OpenSUSE102.zip
1     341204 Aug 14  2014 OpenSUSE103.zip
1     359639 Aug 14  2014 OpenSUSE110.zip
1     416886 Aug 14  2014 OpenSUSE111.zip
1     527383 Aug 14  2014 OpenSUSE113.zip
1     605911 Aug 14  2014 OpenSUSE114.zip
1     672487 Aug 14  2014 OpenSUSE122.zip
1     693843 Aug 14  2014 OpenSUSE123.zip
1     721025 Aug 14  2014 OpenSUSE131.zip
1     620473 Aug 14  2014 OpenSuSE121.zip
1     485406 Aug 14  2014 openSUSE112.zip
```

./Linux/RedHat

```
19     646 Aug 14  2014 x64
19     646 Aug 14  2014 x86
```

./Linux/RedHat/x64

```
1     363080 Aug 14  2014 RedHat50.zip
1     368280 Aug 14  2014 RedHat51.zip
```

```
1      405164 Aug 14  2014 RedHat510.zip
1      371243 Aug 14  2014 RedHat52.zip
1      377464 Aug 14  2014 RedHat53.zip
1      389626 Aug 14  2014 RedHat54.zip
1      393650 Aug 14  2014 RedHat55.zip
1      398169 Aug 14  2014 RedHat56.zip
1      400124 Aug 14  2014 RedHat57.zip
1      402631 Aug 14  2014 RedHat58.zip
1      404656 Aug 14  2014 RedHat59.zip
1      614783 Aug 14  2014 RedHat60.zip
1      628256 Aug 14  2014 RedHat61.zip
1      634384 Aug 14  2014 RedHat62.zip
1      643195 Aug 14  2014 RedHat63.zip
1      653442 Aug 14  2014 RedHat64.zip
1      673408 Aug 14  2014 RedHat65.zip
```

./Linux/RedHat/x86

```
1     356003 Aug 14  2014 RedHat50.zip
1      360889 Aug 14  2014 RedHat51.zip
1      391082 Aug 14  2014 RedHat510.zip
1      363548 Aug 14  2014 RedHat52.zip
1      369260 Aug 14  2014 RedHat53.zip
1      377998 Aug 14  2014 RedHat54.zip
1      381774 Aug 14  2014 RedHat55.zip
1      398169 Aug 14  2014 RedHat56.zip
1      386104 Aug 14  2014 RedHat57.zip
```

```
  1     388559 Aug 14  2014 RedHat58.zip
  1     390461 Aug 14  2014 RedHat59.zip
  1     610104 Aug 14  2014 RedHat60.zip
  1     623267 Aug 14  2014 RedHat61.zip
  1     629121 Aug 14  2014 RedHat62.zip
  1     637224 Aug 14  2014 RedHat63.zip
  1     630896 Aug 14  2014 RedHat64.zip
  1     649151 Aug 14  2014 RedHat65.zip
```

./Linux/Ubuntu

```
  18     612 Sep 18 23:04 x64
  18     612 Mar 29  2015 x86
```

./Linux/Ubuntu/x64

```
 1     607331 Aug 14  2014 Ubuntu10044.zip
  1     648266 Aug 14  2014 Ubuntu1010.zip
  1     711829 Aug 14  2014 Ubuntu1104.zip
  1     732281 Aug 14  2014 Ubuntu1110.zip
  1     902974 Aug 14  2014 Ubuntu12044.zip
  1     711523 Aug 14  2014 Ubuntu1210.zip
  1     747224 Aug 14  2014 Ubuntu1304.zip
  1     861852 Aug 14  2014 Ubuntu1310.zip
  1     927103 Sep 18 23:04 Ubuntu1404.zip
  1     311048 Aug 14  2014 Ubuntu610.zip
  1     322102 Aug 14  2014 Ubuntu704.zip
  1     322958 Aug 14  2014 Ubuntu710.zip
  1     338761 Aug 14  2014 Ubuntu8044.zip
  1    410502 Aug 14  2014 Ubuntu810.zip
  1    523253 Aug 14  2014 Ubuntu904.zip
  1    598615 Aug 14  2014 Ubuntu910.zip
```

./Linux/Ubuntu/x86

```
 1    584694 Aug 14  2014 Ubuntu10044.zip
  1    623111 Aug 14  2014 Ubuntu1010.zip
  1    689118 Aug 14  2014 Ubuntu1104.zip
  1    706971 Aug 14  2014 Ubuntu1110.zip
  1    874175 Aug 14  2014 Ubuntu12044.zip
  1    691304 Aug 14  2014 Ubuntu1210.zip
  1    722911 Aug 14  2014 Ubuntu1304.zip
  1    833346 Aug 14  2014 Ubuntu1310.zip
  1    860071 Mar 29  2015 Ubuntu1404.zip
  1    298760 Aug 14  2014 Ubuntu610.zip
  1    332330 Aug 14  2014 Ubuntu704.zip
  1    324667 Aug 14  2014 Ubuntu710.zip
  1    337231 Aug 14  2014 Ubuntu8044.zip
  1    384262 Aug 14  2014 Ubuntu810.zip
  1    500193 Aug 14  2014 Ubuntu904.zip
  1    575041 Aug 14  2014 Ubuntu910.zip
```

./Mac

```
  3    102 Sep 18 23:04 10.10
  3    102 Oct 28 22:09 10.11
  3    102 Aug 14  2014 10.5
  4    136 Aug 14  2014 10.6
  4     136 Aug 14  2014 10.7
  3     102 Aug 14  2014 10.8
  3     102 Aug 14  2014 10.9
```

./Mac/10.10

```
10     340 Dec  3 15:14 x64
```

./Mac/10.10/x64

```
1     708106 Sep 18 23:04 Yosemite_10.10.2_14C1514.zip
 1     710123 Sep 18 23:04 Yosemite_10.10.3_14D131.zip
 1     710074 Sep 18 23:04 Yosemite_10.10.3_14D136.zip
 1     712410 Sep 18 23:04 Yosemite_10.10.4_14E46.zip
 1     574189 Dec  3 15:14 Yosemite_10.10.5_14F1021.zip
 1     709131 Sep 18 23:04 Yosemite_10.10.5_14F27.zip
 1     708559 Sep 18 23:04 Yosemite_10.10_14A389.zip
 1     708559 Sep 18 23:04 Yosemite_10.10_14B25.zip
```

./Mac/10.11

```
1     745234 Oct 28 22:09 ElCapitan_10.11_15A284.zip
```

./Mac/10.5

```
9     306 Aug 14  2014 x86
```

./Mac/10.5/x86

```
 1     1250296 Aug 14  2014 Leopard_10.5.3_Intel.zip
 1     1251524 Aug 14  2014 Leopard_10.5.4_Intel.zip
 1     1249407 Aug 14  2014 Leopard_10.5.5_Intel.zip
 1     1254176 Aug 14  2014 Leopard_10.5.6_Intel.zip
 1     1152413 Aug 14  2014 Leopard_10.5.7_Intel.zip
```

```
1     1154019 Aug 14  2014 Leopard_10.5.8_Intel.zip
1     1249320 Aug 14  2014 Leopard_10.5_Intel.zip
```

./Mac/10.6

```
10      340 Aug 14  2014 x64
11      374 Mar 29  2015 x86
```

./Mac/10.6/x64

```
1     1078398 Aug 14  2014 SnowLeopard_10.6.1_AMD.zip
1     1079186 Aug 14  2014 SnowLeopard_10.6.2_AMD.zip
1     1078169 Aug 14  2014 SnowLeopard_10.6.4_AMD.zip
1     1081130 Aug 14  2014 SnowLeopard_10.6.5_AMD.zip
1     1081198 Aug 14  2014 SnowLeopard_10.6.6_AMD.zip
1     1088625 Aug 14  2014 SnowLeopard_10.6.7_AMD.zip
1     1094384 Aug 14  2014 SnowLeopard_10.6.8_AMD.zip
1     1078398 Aug 14  2014 SnowLeopard_10.6_AMD.zip
```

./Mac/10.6/x86

```
1     1081174 Aug 14  2014 SnowLeopard_10.6.1_Intel.zip
1     1082113 Aug 14  2014 SnowLeopard_10.6.2_Intel.zip
1     1078938 Mar 29  2015 SnowLeopard_10.6.3_Intel.zip
1     1081377 Aug 14  2014 SnowLeopard_10.6.4_Intel.zip
1     1084183 Aug 14  2014 SnowLeopard_10.6.5_Intel.zip
1     1083934 Aug 14  2014 SnowLeopard_10.6.6_Intel.zip
1     1091880 Aug 14  2014 SnowLeopard_10.6.7_Intel.zip
1     1097540 Aug 14  2014 SnowLeopard_10.6.8_Intel.zip
1     1081174 Aug 14  2014 SnowLeopard_10.6_Intel.zip
```

./Mac/10.7

```
8     272 Aug 14  2014 x64

8     272 Aug 14  2014 x86
```

./Mac/10.7/x64

```
1     1205691 Aug 14  2014 Lion_10.7.1_AMD.zip
1     1206992 Aug 14  2014 Lion_10.7.2_AMD.zip
1     1206610 Aug 14  2014 Lion_10.7.3_AMD.zip
1     1209528 Aug 14  2014 Lion_10.7.4_AMD.zip
1     1210575 Aug 14  2014 Lion_10.7.5_AMD.zip
1     1205879 Aug 14  2014 Lion_10.7_AMD.zip
```

./Mac/10.7/x86

```
1     1222054 Aug 14  2014 Lion_10.7.1_Intel.zip
1     1223954 Aug 14  2014 Lion_10.7.2_Intel.zip
1     1223590 Aug 14  2014 Lion_10.7.3_Intel.zip
1     1226080 Aug 14  2014 Lion_10.7.4_Intel.zip
1     1227728 Aug 14  2014 Lion_10.7.5_Intel.zip
1     1221746 Aug 14  2014 Lion_10.7_Intel.zip
```

./Mac/10.8

```
8     272 Aug 14  2014 x64
```

./Mac/10.8/x64

```
1     1238687 Aug 14  2014 MountainLion_10.8.1_AMD.zip
1     1239024 Aug 14  2014 MountainLion_10.8.2_AMD.zip
1     1238934 Aug 14  2014 MountainLion_10.8.3_AMD.zip
1     1240705 Aug 14  2014 MountainLion_10.8.4_12e55_AMD.zip
1     1251778 Aug 14  2014 MountainLion_10.8.5_12f37_AMD.zip
1     1251712 Aug 14  2014 MountainLion_10.8.5_12f45_AMD.zip
```

./Mac/10.9

```
12     408 Dec  3 15:14 x64
```

./Mac/10.9/x64

```
 1     8365062 Oct 14  2014 10.9.5.64bit.symbol.dsymutil

 1     1279883 Oct 14  2014 10.9.5.64bit.vtypes

 1     1333931 Aug 14  2014 Mavericks_10.9.1_AMD.zip

 1     1334545 Aug 14  2014 Mavericks_10.9.2_13C1021.AMD.zip

 1     1334596 Aug 14  2014 Mavericks_10.9.2__13C64.AMD.zip

 1     1335250 Aug 14  2014 Mavericks_10.9.3_AMD.zip

 1     1335990 Aug 14  2014 Mavericks_10.9.4_AMD.zip

 1     1291868 Dec  3 15:14 Mavericks_10.9.5_13F1077_AMD.zip

 1    1336065 Oct 21  2014 Mavericks_10.9.5_AMD.zip
```