Hunting Mac OS X rootkit with Memory Forensics

Kyeongsik Lee[1], Jinkook Kim[2], Hyungjoon Koo[2]

[1]Defense Cyber Warfare Technology Center, Agency for Defense Development, Sonpa P.O Box 132, Seoul, Republic of Korea
[2]Center for Information Security Technologies (CIST), Korea University, Anam-Dong, Seongbuk-Gu, Seoul, Republic of Korea

Abstract

The anti-virus product for Mac OS X has emerged recently, reflecting its growing popularity. While cyber warfare among countries has focused on Windows operating system so far, modern APT also aims at Mac OS X. In the case of Tibet attack and Flashback, Mac OS becomes no longer safe zone. Most elaborate exploits have started taking advantage of common vulnerabilities available on cross-platform. The rootkit for Mac OS X called *rubilyn* has been found lately, whose open source enables newbie hackers to create other variations with ease. Suppose that clever rootkit would successfully penetrate into end users. Then it could perform a variety of malicious activities without any suspicious event recognition and even incapacitate anti-virus solution for Mac. Some vendors have already combined their engines with memory forensic technique for Windows platform, which allows them to discover unpacked malware and/or rootkits by looking into memory investigation. This advanced technique improves detection rates of malware to make an attempt to bypass known signatures. The research on memory forensics for Mac OS X has been conducted over the last couple of years. This paper covers characteristics of a prevalent Mac malware and diverse analysis techniques based on the ongoing project, *volafox*, including general system information view, hidden process detection, hidden KEXT detection, System Call Table and Mach Trap Table hooking detection and so forth. Furthermore, we present efficient rootkit detection method using memory forensic techniques for Mac, getting over anti-virus bypass.

Keywords: Mac OS X, Memory Forensics, *volafox* project, Rootkit Detection.

1. Introduction

While malware authors mainly targeted Microsoft Windows family, lately they have widened into diverse OS targets including Mac OS X. A good example of APT (Advanced Persistent Threat) would be Tibet attack, which aimed at Mac OS X. This means Mac OS X is not safe zone any more. [1]

Malware at the early stage of Mac OS X did not have built-in exploits and/or rootkits. It used simple technique that disguised oneself as legitimate application or file. However, modern Mac malware often installs rootkits and then makes an attempt to hide a specific process, to tamper a process ownership, to capture key strokes, and to control audio devices and so forth. These rootkits has a device driver, called KEXT or Kernel Extension.

In particular, *rubilyn* rootkit publicly released its source code in 2012, which supports built-in rootkit features. [2] This allows a script kiddie to create his/her own rootkit variants by simple modification of known source. However, current anti-virus solutions for Mac depend largely on signature-based detection. Thus they have limitations to detect customized rootkits because they often manipulate kernel memory and pass wrong outputs to applications.

One of most effective method to detect rootkits is memory forensics technique, which allows investigators to analyze logical memory structure and to extract forensically meaningful data. This technique helps incident handling because physical memory is the only place to be unlikely to conceal malware itself.

This paper presents useful techniques with Mac OS X memory forensics, especially with regard to the

[1]*Corresponding author.* Fax: +82 2 403 3512 (15826).
E-mail addresses: rapfer@gmail.com (KS, Lee).

[2]E-mail addresses: proneer@gmail.com(JK. Kim), kevinkoo001@gmail.com (HJ. Koo)

concealment and detection technique of rootkits.

2. Methods for Mac OS X Memory Analysis

There are three steps for Mac memory analysis: extracting symbol address within kernel image, finding kernel base address considered KASLR and extracting useful artifacts in memory. This section illustrates each phase in sequence.

2.1. Extracting Symbol Address within Kernel Image

The first step of Mac memory analysis is to obtain virtual address from kernel symbols. The file *mach_kernel*, Mac OS X kernel image, follows Universal Binaries file format to support multiple platforms in Mac OS X Lion or earlier. On the other hand, the kernel in Mac OS Mountain Lion or later follows Mach-O file format, which only supports 64 bits.

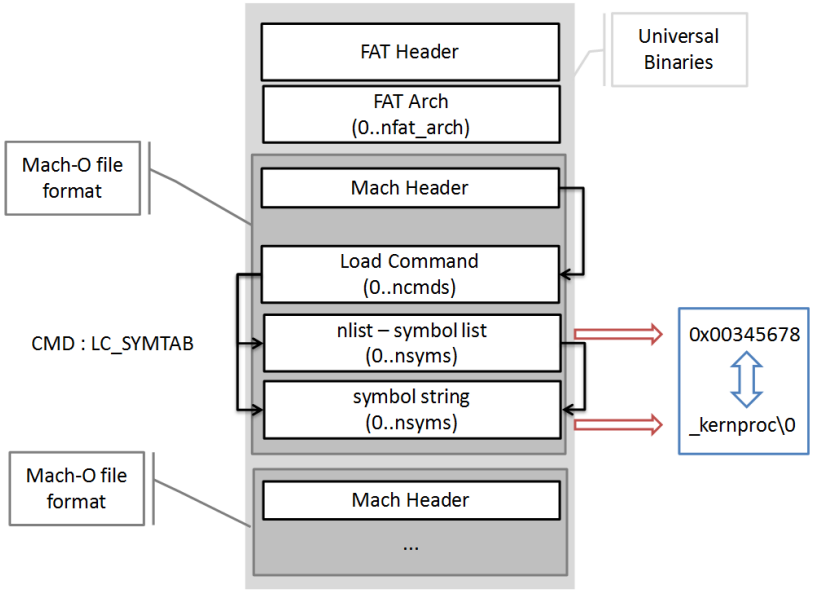

Figure 1. Universal Binaries & Mach-O File Format [3]

Note that it is essential to identify running CPU architecture and corresponding symbol tables because each Mach-O file has its own kernel symbols. Based upon Universal Binaries and Mach-O file format from Apple official documents, it is able to extract symbol information. As **Figure 1** shows, the location and the size of symbol tables can be found with *LC_SYMTAB* from *Load Command*. Using obtained table information, symbol name and virtual address of each symbol can be extracted, which helps the data structure analysis in memory.

2.2. Finding Kernel Base Address considered KASLR

Until Mac OS X Lion or earlier, the loaded kernel page table, *IdlePDPT* for 32 bit kernel or *IdlePML4* for 64 bit kernel, allowed page table configuration and memory analysis because symbol address directly maps to physical address. However, Apple has hardened the mechanism applying KASLR technique (known as Kernel Address Space Layout Randomization) in Mac OS X Mountain Lion or later so that it could make rootkits difficult to predict the address of significant objects for operating system. [4] This also indicates that previous memory analysis tools for Mac might be problematic for appropriate data interpretation due to the starting address alteration of the page table. To resolve this matter, entire symbol addresses need the adjustment by looking for kernel base address and adding it into each symbol address. One of the Mac OS X memory analysis tools, *volafox*, identifies kernel base address as following:

(1) Find *_lowGlo*, one of kernel symbols. The *_lowGlo* is the virtual address of *lowvector* structure whose signature is '*Catfish* '. Note that the last character of the signature has a single space.
(2) Search for *'Catfish '* from the page block in physical memory until the string is found.
(3) Calculate the difference by subtracting the address found '*Catfish* ' signature from *_lowGlo* kernel symbol address, which eventually discover kernel base address. Note that kernel symbol address should be converted into 32 bits because Mac OS Mountain Lion or higher represents all virtual addresses in 64 bits only. It can be obtained simply by taking the remainder to divide the 64 bits value by 0xFFFFFF80.

After kernel base address is found, the physical address of page table can be identified as following steps:
(1) Configure the page table in *BootPML4* symbol address.
(2) Find the physical address with the page table at the first step using the value adding *IdlePML4* to kernel base address above.
(3) Find the address of kernel page table by reading 8 bytes in the physical address at the second step.

Once page table address is identified, we move on extracting artifacts in memory.

2.3. Extracting Useful Artifacts in Memory

Once page table and kernel base address are identified, it is ready for kernel structure analysis. The information which kernel symbols point to can be accessible with physical address, using page table by adding symbol virtual address extracted from kernel image to kernel base address. For instance, the following phase shows how to obtain *machine_info* structure.

(1) The symbol address of *machine_info* can be found in kernel symbol list.
(2) In case of OS X Mountain Lion, symbol address should be adjusted by adding altered kernel base address due to KASLR.
(3) Using page table from **section 2.2**, it allows virtual address to convert into physical address.
(4) The information can be interpreted from physical address in a proper manner according to *machine_info* structure.

3. Rootkit Technique and Detection

To begin with, Mac OS X rootkits are highly likely to take advantage of techniques based on *Nix rootkit. However, their techniques are getting more complicated and elaborated. This section covers how to detect Mac rootkit along with its features.

3.1. Process Hiding with DKOM

Process hiding technique is one of fundamental features which general rootkit has implemented. This feature helps to make an attempt to conceal a particular process with various techniques. In this section, we focus on process object manipulation, which is well known for DKOM or Direct Kernel Object Manipulation.

Mac OS X has two major components: Mach, BSD. Each component has its own role to manage process as following: [5]

- Mach: virtual memory management
- BSD: process ownership and process management

In order to hide rootkit process itself, it performs either manipulation of doubly linked list in *proc* structure or removal of *proc* structure pointer address for particular process in hash table. If the process information in *proc* structure is tampered, it is no longer identifiable from *ps* or *top* command to list current processes.

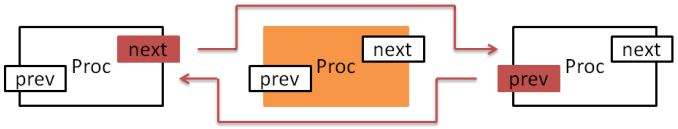

Figure 2. Process Hiding with DKOM (Direct Kernel Object Manipulation)

Figure 2 illustrates that general system commands would not show a hidden process because the process structure pointer has been manipulated by unlinking back and forth. As you might imagine, the hidden process with the pointer alteration is running without any trouble for sure. In this case, it is necessary to have another detection technique.

At first, manipulated kernel object can be identifiable with other structure information related to process structure. Mac OS X process management has divided into two parts in aforementioned components: process management and virtual memory management. The *task* structure in Mach components maintains one-to-one mapping with *proc* structure and each connects to virtual memory management structure in **Figure 3**. In other words, hidden process can be revealed if *task* structure tracks down *proc* structure in reverse.

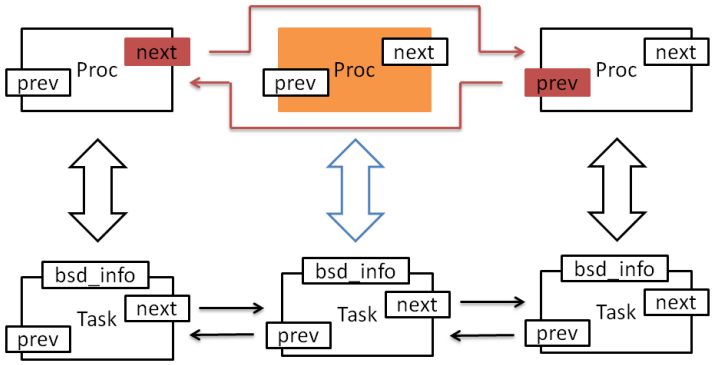

Figure 3. Discovering hidden process with *task* structure

Memory analysis tool makes it possible to extract all *proc* structures with the *bsd_info* pointer pointing to *proc* structure, by tracing *task* structure in kernel symbol. Compared to the result from *proc* structure trace, we end up with discovering a hidden process.

The other way is to use hash table in *proc* structure. Hash table is designed to allow applications to acquire process information quickly, thus it contains the pointer of each structure. With this table, it is feasible to detect hidden rootkit with doubly linked list manipulation. Note that this technique might be less useful than the detection using *task* structure because a hidden process can be normally running after the pointer in hash table has been eliminated. [6]

3.2. KEXT Hiding with DKOM

Mac OS X provides developers with *I/O Kit* Framework which allows them to add modules in need at XNU kernel. [7] The kernel module working on *I/O Kit* is called KEXT or Kernel Extensions. KEXT has the privilege to directly get access to kernel memory space. Hence it should be loaded with root privilege. With KEXT, private methods identified through reverse engineering can be used as well as exported methods in kernel. Common rootkit developers often make use of KEXT file to handle kernel memory in handy. Most forensic investigators attempt to detect rootkit with suspicious KEXT identification using *kextstat*, one of Apple system commands.

Rootkit developers employ two hiding techniques from the detection above. One is to return the manipulated outcome of *kextstat* to users by hooking system call. With hooking the *write_nocancel* function of system calls, a printed message in terminal could be easily forged. The other is to fabricate kernel object information in order to conceal rootkit KEXT. [8] By KEXT structure manipulation, rootkit writers successfully hide particular rootkit

KEXT from Apple system commands. However, this technique is not in use any more since it causes kernel panic in Mac OS X Snow Leopard or later. Nonetheless, it is obvious that rootkit authors keep making an effort to manipulate command results.

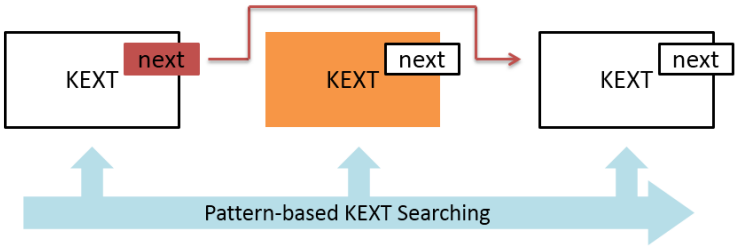

Figure 4. KEXT hiding technique and hidden KEXT discovery

Figure 4 demonstrates one of the best ways based upon specific pattern to discover hidden KEXT. The pattern is defined as characteristics which each field in KEXT structure contains. Then KEXT search has to be exhaustively performed within the memory space of which the structure has been loaded. If any pattern corresponding to pre-defined feature is discovered while searching process, put it on the rootkit candidate list for further examination. This kind of data extraction is called carving technique. **Table 1** shows KEXT search patterns in 64 bit kernel.

Table 1. KEXT Search Patterns in Virtual Address Area (from 0xffffff7f00000000 to 0xffffff8100000000)

Field	Pattern	Size (Bytes)
INFO	1	4
KID	0xFFFF0000 below	4
Reference List	0xffffff7f00000000 ~ 0xffffff8100000000	8
Base Address	0xffffff7f00000000 ~ 0xffffff8100000000	8
Start Function Pointer	0xffffff7f00000000 ~ 0xffffff8100000000	8
Stop Function Pointer	0xffffff7f00000000 ~ 0xffffff8100000000	8
KEXT name	String(ASCII)	64
Version	String(ASCII)	64

When scanning physical memory address with a series of patterns within unique KEXT field, it is possible to extract all hidden KEXT structures whatever technique has been employed in practice. Keep in mind that you need to verify the loaded KEXT list because this technique draws the unloaded ones as well.

3.3. Network Session Hiding

The BSD component of XNU kernel in Mac OS X is responsible for network session management. The commands for network query such as *netstat* get access to session information with *inpcbinfo* structure. Mac creates *inpcb* structure and connects it to each other whenever a new network session is made.

Rootkit authors are able to hide network sessions if they could alter *inpcb* structure. The hiding technique is the same with process one by doubly linked list manipulation. Once forged, system network commands are unable to expose manipulated objects. If this network hiding technique is combined with aforesaid process hiding over attack, then system administrator have no choice but to be exploited without any perception.

It is available to use hash table against network session hiding. The BSD component holds hash table in *inpcbinfo* structure in order to access to *inpcb* structure with better efficiency. For example, general web service would create a large number of network sessions, which generates high overhead to collect all network information from doubly

linked *inpcb*. Therefore, deploying hash table allows to avoid overhead and to collect the information with pointers for each *inpcb* structure. (**Figure 5**) This technique helps to discover hiding network sessions. Moreover, if an attacker tries to remove network information in the table on purpose, network session would be no longer valid. This is different part from process hiding which hash table does not influence on.

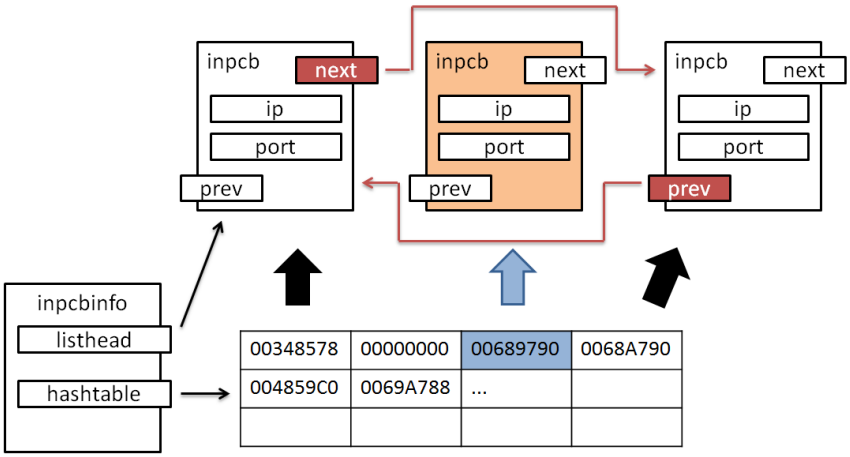

Figure 5. Hidden Network Sessions Discovery

3.4. System Call Table / Mach Trap Table Hooking

Likewise other Unix system, Mac OS uses a function table, which allows applications to access kernel resources. As already mentioned, Mac OS consists of Mach and BSD components and each component provides function table for kernel system resource access. The table in BSD components is called System Call Table, whereas the one in Mach components is called Mach Trap Table. Of the two, System Call Table manipulation is system call hooking. There are two techniques for function hooking: one is to manipulate system call (**Figure 6**) and the other is to manipulate function code directly. Since the former offers more convenient and safer way to implement than the latter, all known rootkits have adapted the former technique so far.

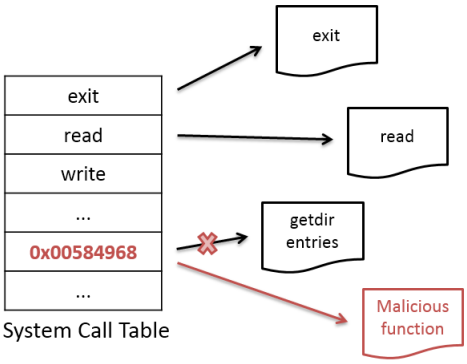

Figure 6. System Call Table Hooking

If predefined number for system call is negative, Mach Trap Handler is executed instead of BSD system call. Mach Trap Table could be forged by function pointer and/or code manipulation. However the case that rootkit directly forges Mach Trap Table has yet to be reported because most functions in Mach Trap Table control Mach Components structure as well as most system commands for management acquire information from BSD structure.

The method to detect table hooking can be narrowed down to the two: comparing System Call Table with Kernel Symbol List in kernel image and using KEXT information.

The first method is to compare the calls in System Call Table and the ones in Kernel Symbol List. (**Figure 7**) It is possible to check if there is any system call manipulation because System Call Table and Mach Trap Table are loaded into virtual address within symbol information. This technique verifies entire calls in System Call Table and handlers in Mach Trap Table because both tables hold virtual address stored Kernel Symbol List unless those are manipulated. Make sure that Mac memory analysis tool contains its own symbol information for each kernel version since the information might be wrong in case of being corrupted in kernel. Note that the system call, *fsgetpath*, does not have any symbol in kernel thus it has been seemingly hooked at all times. [9]

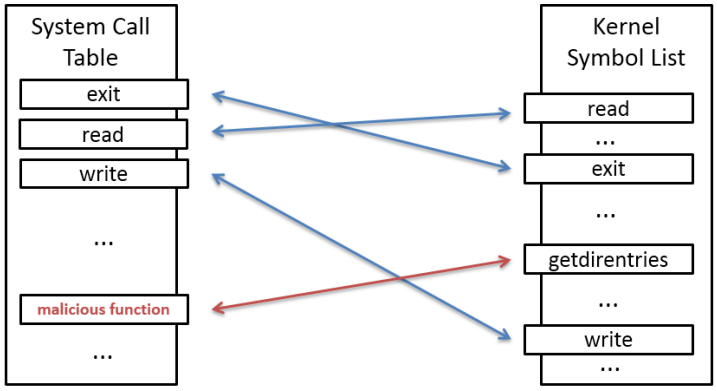

Figure 7. Finding System Call Hooking using Kernel Symbol List

Another method is to use KEXT structure loaded into memory. This structure maintains KEXT name, loaded virtual address, allocated size and so forth. If System Call and Mach Trap Handler are on normal status, they point to *__kernel__* of KEXT memory space, where KEXT ID is 0. In case of System Call hooking, it is feasible to check the validation since different area in KEXT would be pointed to. As **Figure 8** shows, the system call *getdirentries* points to an abnormal position.

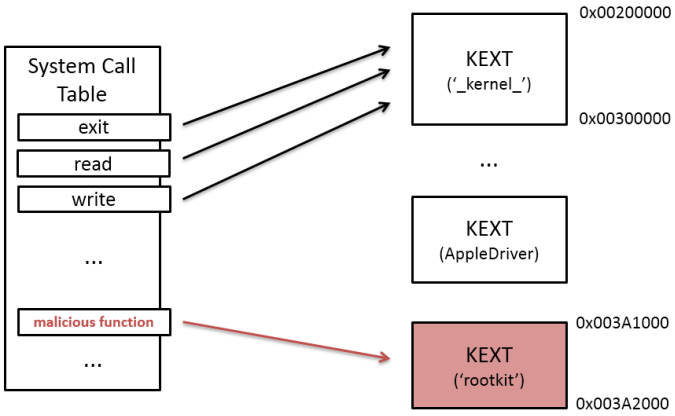

Figure 8. System Call Hooking Detection comparing with KEXT Memory Area

3.5. Process Privilege Escalation

Normal process is inherited by the privilege of user or parent process which executes it. The *Proc* structure in BSD Components administers UID and GID field of each process. The fields in *Proc* structure are used only when storing process information. The *cred* structure controls a process privilege in practice. As **Figure 9** shows, once the privilege information in *cred* structure has been changed, the process would be running with altered one.

Rootkit takes advantage of this feature in order to get privilege escalation for a certain process. This means target

process is able to get access to any data with root privilege because of unwanted forgery.

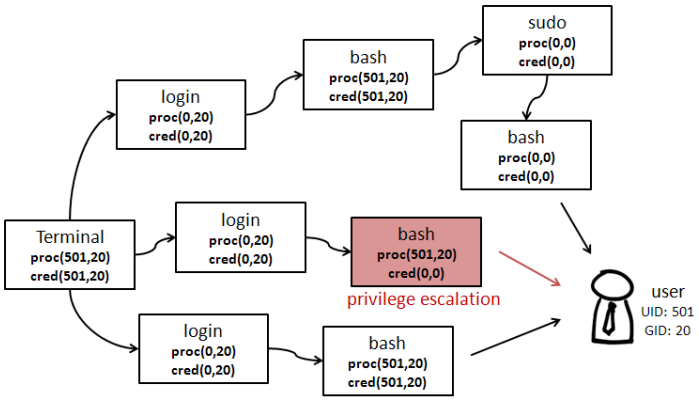

Figure 9. Process Privilege Escalation

To detect the process with escalated privilege by rootkit, the relationships among processes acquired from memory should be considered. Besides, the UID and GID of *proc* structure should be compared with those of *cred* structure.

If the child process whose parent has user privilege has been running with root one, this might be questionable process exploited by rootkit. Note that care must be taken to avoid false positive when child process has been created by parent process which *sudo* command executed and/or the process with *setuid* configuration has been running, because those cases are legitimate privilege escalation.

Rootkit sometimes manipulate merely *cred* structure to evade the detection of privilege forgery. In other words, the UID and GID in *proc* structure remains intact, which rootkit pretends to be a normal process. Therefore it is detectable to have illegal privilege escalation by UID and GID comparison between the two aforementioned structures.

3.6. *TrustedBSD* Framework analysis

Mac OS X supports two types of access control: Discretionary Access Control or DAC and Mandatory Access Control or MAC. DAC is that each user has one's own permission for files with Access Control List or ACL. In Mac OS X, file system stores the ACL of each file and operating system checks the access permission with the ACL. MAC uses *TrustedBSD* Framework in BSD Components. [10] The framework provides users with access control environment based upon security policy. *TrustedBSD* allows to limit particular process resource (socket or IPC) and/or a file as well as to get object-level security other than DAC. *TrustedBSD* renames security policy to MAC policy. However, it does nothing if not registered in the framework through KEXT. A developer need to develop KEXT and register a handler to handle operation vector in MAC policy. The framework provides approximately 600 operation vectors in the form of *mpo_object_operation_call* on Mac OS X Mountain Lion. For instance, if KEXT registers a handler for *mpo_mount_check_mount* operation vector then kernel will call a registered handler when *mount* system call is called with *mount* object.

Apple also supports *Sandbox* to prevent one application from another or system access. [11] *Sandbox* allows application to access designated resources based on the description in an application. Presently *Sandbox* can be only applied to the applications purchased and downloaded from *App-Store*. It also uses *TrustedBSD* Framework for the purpose of handling application resource access. (**Figure 10**)

```
--------------------------------------------------------------------------------
Name: Sandbox, Full Name: Seatbelt sandbox policy
Loadtime : unknown, Runtime: MPC_RUNTIME_FLAG_REGISTERED
--------------------------------------------------------------------------------
Entrypoint                      Virtual Address Physical Address
mpo_cred_check_label_update_execve 0xFFFFFF7F808370CB       0x0533F0CB
mpo_cred_check_label_update     0xFFFFFF7F8082EAF8       0x05376AF8
mpo_cred_label_associate        0xFFFFFF7F8082EB24       0x05376B24
mpo_cred_label_destroy          0xFFFFFF7F8082EB4B       0x05376B4B
mpo_cred_label_update_execve    0xFFFFFF7F808370F4       0x0533F0F4
mpo_cred_label_update           0xFFFFFF7F8082EB57       0x05376B57
mpo_file_check_fcntl            0xFFFFFF7F8082EB89       0x05376B89
mpo_file_check_set              0xFFFFFF7F8082EBA1       0x05376BA1
mpo_mount_check_fsctl           0xFFFFFF7F8082EBB0       0x05376BB0
mpo_mount_check_mount           0xFFFFFF7F8082EBBF       0x05376BBF
mpo_mount_check_remount         0xFFFFFF7F8082EBD6       0x05376BD6
mpo_mount_check_umount          0xFFFFFF7F8082ED7B       0x05376D7B
mpo_policy_init                 0xFFFFFF7F8082EF20       0x05376F20
mpo_policy_initbsd              0xFFFFFF7F8082EFB2       0x05376FB2
mpo_policy_syscall              0xFFFFFF7F8082EFEB       0x05376FEB
```

Figure 10. Sandbox Policy

One of Mac OS X reversers has written rootkit based on *TrustedBSD* Framework. [12] This rootkit is called *rex the wonder dog* with KEXT. Once loaded KEXT, new MAC policy is registered. In addition, it registers the handler for operation vector named *mpo_proc_check_get_task* so that the process, *xyz*, could provide the shell with root privilege when calling *task_for_pid()*. **Figure 11** illustrates the simple diagram of rootkit flow.

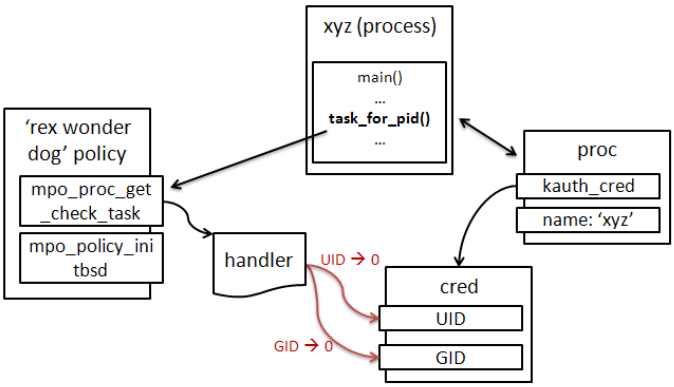

Figure 11. *rex the wonder dog* rootkit flow diagram

Andrew Case, *volatility* co-developer, has written the module to be able to analyze MAC policy from acquired memory image, which helps to detect *TrustedBSD* Framework abuse. [13] Since the framework maintains normal MAC policy, it should be confirmed with KEXT information together. If any MAC policy has been discovered other than *TMSafetyNet* (Safety Net for Time Machine), *Quarantine* (Quarantine Policy) and *Sandbox* (Seatbelt Sandbox Policy), it might be suspicious because those three policies are registered by default from initial Mac OS X installation process. The KEXT can be found through MAC policy registered by *TrustedBSD* Framework and virtual function address configured by operation vector. Keep in mind that KEXT can be used in a legitimate manner such as anti-virus product.

4. Case Study: *rubilyn* rootkit Analysis

4.1. *Rubilyn* rootkit

Rublyn rootkit consists of KEXT and the console to control it, whose source code has been open to the public at *Null Security*. Yet no case has been reported to distribute malware with the accessible code. However, chances are still high to misuse it with diverse variants. This section presents what features this rootkit have and how to check each one in memory.

The following shows what to perform as main features in *rubilyn*.

- Hiding a desired file
- Hiding a desired process which has a particular PID
- Manipulating the output of system commands such as *kextstat*, *ps* and so forth
- Hiding a network communication for a particular message with a backdoor

In this paper, we acquired the memory image of an infected machine which all *rubilyn* features were on. Memory analysis was done by *volafox* which allows investigators to support all concealment techniques described in **Section 3**. [14]

4.2. Memory Analysis of an Infected System

Before going into deep memory analysis, we checked basic information of the system with the following command. Using the options such as *system_profiler* and *uname* from *volafox*, it was available to identify the infected system information including the kernel version, the size of memory, and the number of possible CPUs and so on. (**Figure 12**)

 # python volafox –i [memory image] –o system_profiler

```
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$ python vol.py -i rubilyn.flat -o system_profiler
[+] Mac OS X Basic Information
 [-] Darwin kernel Build Number: 11B26
 [-] Darwin Kernel Major Version: 11
 [-] Darwin Kernel Minor Version: 1
 [-] Number of Physical CPUs: 1
 [-] Size of memory in bytes: 2147483648 bytes
 [-] Size of physical memory: 2147483648 bytes
 [-] Number of physical CPUs now available: 1
 [-] Max number of physical CPUs now possible: 1
 [-] Number of logical CPUs now available: 1
 [-] Max number of logical CPUs now possible: 1
 [-] Last Hibernated Sleep Time: Thu Jan 01 00:00:00 1970 (GMT +0)
 [-] Last Hibernated Wake Time: Thu Jan 01 00:00:00 1970 (GMT +0)
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$
```

Figure 12. System Profiler with *volafox*

It is commonplace to tamper System Call Table and/or Mach Trap Table, therefore the status of these tables should be examined from the beginning. If *volafox* detects a suspicious hooking, it displays the message, *maybe hooked*. In this case, the following command with *grep* brought four system call hooks. (**Figure 13**)

 # python volafox –i [memory image] –o [systab/mtt] | grep hooked

```
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$ python vol.py -i rubilyn.flat -o systab | grep "hooked"
222         8   0    0            0xFFFFFF7F80BFF41D 0xFFFFFF7F80BFF41D 0xFFFFFF80005CC110      0x00000000      1       32 Maybe hooked
344         4   0    0            0xFFFFFF7F80BFF2EE 0xFFFFFF7F80BFF2EE 0xFFFFFF80005CC0D0      0x00000000      6       16 Maybe hooked
397         3   0    0            0xFFFFFF7F80BFFA7E 0xFFFFFF7F80BFFA7E 0xFFFFFF80005CC0C0      0x00000000      6       12 Maybe hooked
427         4   0    0            0xFFFFFF8000300710 0xFFFFFF8000300710 0xFFFFFF80005CC240      0x00000000      6       20 Maybe hooked
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$ python vol.py -i rubilyn.flat -o mtt | grep "hooked"
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$
```

Figure 13. Hooking Detection with System Call and Mach Trap Table

Note that the output in **Figure 13** had system call 427 because no system call information exists in kernel symbol list as mentioned in **Section 3.4**. Thus, this does not necessarily mean it was tampered system call by rootkit. Furthermore, we could anticipate a single module hooking system call because the function pointers of system call were quite close. The header file indicated three system calls as **Figure 14**.

```
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$ egrep -e "222|344|397" /usr/include/sys/syscall.h
#define SYS_getdirentriesattr 222
#define SYS_getdirentries64 344
#define SYS_write_nocancel 397
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$
```

Figure 14. The Name of System Call Hooking

Here is common role for each system call hooking.

- *getdirentriesattr*: This gets file system attributes for multiple directory entries.
- *getdirentries64*: This gets directory entries in a file system independent format for 64bit inode.
- *write_nocancel*: All output from Mac OS X system would be seen to user through this system call.

As shown above, system call hooking will do the followings:

- Manipulating the file list provided by system commands such as *finder* and *ls*
- Manipulating various outputs provided by system commands with system call hooking

The system call hooking pointers helped to determine hooking modules. The *volafox* below showed the KEXT list from dumped memory. (**Figure 15**)

 # python volafox –i [memory image] –o kextstat

```
[+] Kernel Extention List
OFFSET(P)  INFO KID                                          KEXT_NAME          VERSION REFER_COUNT        REFER_LIST
        ADDRESS    SIZE HDRSIZE          START_PTR STOP_PTR
0x314651D0    1  92                                com.atc-nycorp.devmem.kext      3.0.2           0 0xFFFFFF800818A080 0
xFFFFFF7F80C09000   12288       0 0xFFFFFF7F80C0AAFF 0xFFFFFF7F80C0AB4B
0x2E52C7C8    1  91                                com.hackerfantastic.rubilyn        1           0 0xFFFFFF8006A0FF20 0
xFFFFFF7F80BFE000   20480       0 0xFFFFFF7F80BFFE6A 0xFFFFFF7F80BFFEB6
0x134D1CD0    1  90                              com.apple.driver.AppleHWSensor  1.9.4d0           0 0xFFFFFF8008181EC0 0
xFFFFFF7F80C73000   20480       0 0xFFFFFF7F80C75A97 0xFFFFFF7F80C75AE3
0x11CE64B0    1  89                               com.apple.filesystems.autofs      3.0           0 0xFFFFFF800836DB00 0
xFFFFFF7F80C68000   36864       0 0xFFFFFF7F80C6EFFA 0xFFFFFF7F80C6F046
0x12490020    1  88                                  com.apple.kext.triggers      1.0           1 0xFFFFFF80069C9E00 0
xFFFFFF7F80C63000   20480       0 0xFFFFFF7F80C65BF6 0xFFFFFF7F80C65C42
0x06CE5290    1  87                              com.apple.iokit.IOUserEthernet  1.0.0d1           0 0xFFFFFF80082DD300 0
xFFFFFF7F80C5A000   24576       0 0xFFFFFF7F80C5CC31 0xFFFFFF7F80C5CC7D
0x08326B10    1  86                                com.apple.iokit.IOSurface     80.0           0 0xFFFFFF80082DD500 0
xFFFFFF7F80C48000   73728       0 0xFFFFFF7F80C5005C 0xFFFFFF7F80C500A8
```

Figure 15. The Output for KEXT list with *volafox*

The system command, *kextstat*, had slightly different KEXT list in live system. (**Figure 16**)

```
   83    1 0xffffff7f80c25000 0xe000     0xe000     com.apple.iokit.IOSerialFamily (10.0.5) <7
   84    0 0xffffff7f80c33000 0xe000     0xe000     com.apple.iokit.IOBluetoothSerialManager (
3 1>
   86    0 0xffffff7f80c48000 0x12000    0x12000    com.apple.iokit.IOSurface (80.0) <7 5 4 3
   87    0 0xffffff7f80c5a000 0x6000     0x6000     com.apple.iokit.IOUserEthernet (1.0.0d1) <
   88    1 0xffffff7f80c63000 0x5000     0x5000     com.apple.kext.triggers (1.0) <7 6 5 4 3 1
   89    0 0xffffff7f80c68000 0x9000     0x9000     com.apple.filesystems.autofs (3.0) <88 7 6
   90    0 0xffffff7f80c73000 0x5000     0x5000     com.apple.driver.AppleHWSensor (1.9.4d0) <
victims-MacBook-Pro:/ victim$
```

Figure 16. The Output for KEXT list with System Command, *kextstat*

Compared the first result with the second, we concluded that there was one more KEXT in acquired memory. Both console commands should have resulted in the same output because they followed the single linked list of KEXT structure. The difference certainly came from *write_nocancel* system call hooking by rootkit. This technique could merely impact on the output in terminal but not on kernel objects, thereby led to successfully hide a particular object. In other words, this means simple information extraction from dumped memory could thwart the hooking technique. **Figure 17** presents how to discover a malicious KEXT and dump it from the starting address and the size, which was the rootkit named *com.hackerfantastic.rubilyn*. The following command confirmed our hypothesis that a doubtful rootkit might infect the target system.

 # python volafox –i [memory image] –o kextstat –x [KID]

```
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$ python vol.py -i rubilyn.flat -o kextstat -x 91
[+] Find KEXT: com.hackerfantastic.rubilyn, Virtual Address : 0xFFFFFF7F80BFE000, Size: 20480
[DUMP] FILENAME: com.hackerfantastic.rubilyn-ffffff7f80bfe000-ffffff7f80c03000
[DUMP] Complete.
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$ file com.hackerfantastic.rubilyn-ffffff7f80bfe000-ffffff7f80c03000
com.hackerfantastic.rubilyn-ffffff7f80bfe000-ffffff7f80c03000: Mach-O 64-bit kext bundle x86_64
n0fate@n0fate-ui-MacBook-Pro: ~/volafox$
```

Figure 17. A Malicious KEXT Object Discovery and Dumping

Figure 18 illustrates the IDA Pro, one of the most popular reverse engineering tools, helped to identify functions and system call hooking pointers with ease.

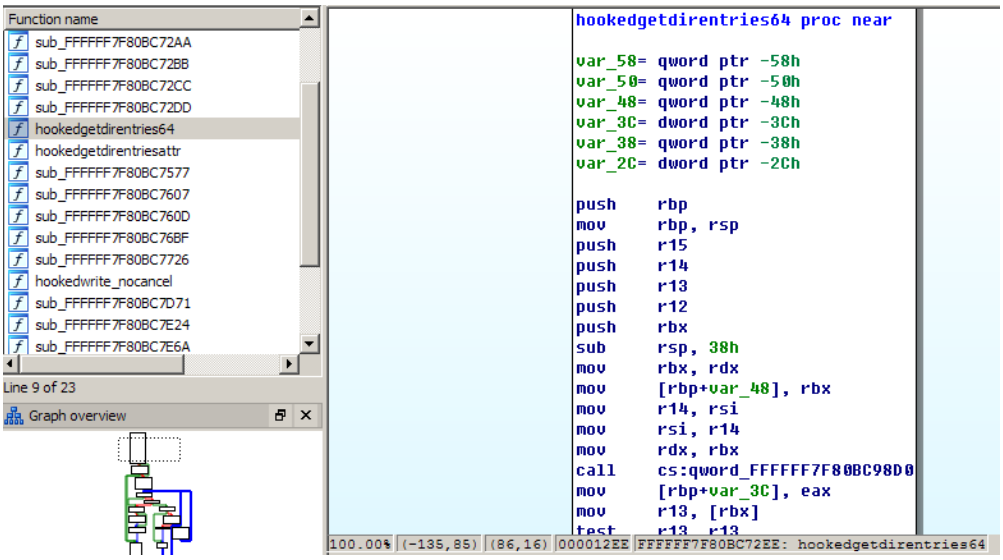

Figure 18. Analysis with IDA Pro

Sometimes *rubilyn* escalates the privilege for a specific process. The *ps* option from *volafox* enabled to recognize the process list and the details of each process.

 # python volafox –i [memory image] –o ps

```
170  116   128   0 AirPort Base Sta      victim(501,20)      (501,20) Wed Apr 17 14:08:30 2013
179  116   255   0         mdworker      victim(501,20)      (501,20) Wed Apr 17 14:09:33 2013
192  125   128   0            login        victim(0,20)        (0,20) Wed Apr 17 14:09:55 2013
194  192   128   0             bash      victim(501,20)        (0,0) Wed Apr 17 14:09:55 2013
212    1   128   0            ocspd   _coreaudiod(0,0)        (0,0) Wed Apr 17 14:11:33 2013
220  194   128   0             sudo        victim(0,0)        (0,0) Wed Apr 17 14:12:29 2013
221  220   128   0 MacMemoryReader        victim(0,0)        (0,0) Wed Apr 17 14:12:29 2013
222    1   128   0        taskgated   _coreaudiod(0,0)        (0,0) Wed Apr 17 14:12:29 2013
228  221   128   0            image        victim(0,0)        (0,0) Wed Apr 17 14:12:31 2013
```

Figure 19. The discovered process which has different privilege of *proc* and *cred* structure in process list

The *bash* process in **Figure 19** had different permission of UID (501) and GID (20) in *process* structure from the permission of UID (0) and GID (0) in *cred* structure. Provided that the owner of this process was general user and *sudo* command was not in use, it must be escalating the permission of *bash* process for rootkit by manipulating *cred*.

Rootkit has a built-in feature to hide information as mentioned earlier. The *tasks* option from *volafox* efficiently revealed a hidden process which tampered *proc* structure with DKOM technique.

 # python volafox –i [memory image] –o tasks

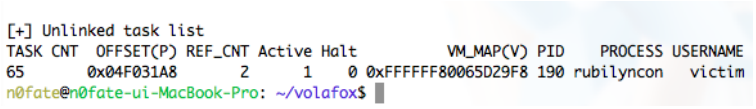

Figure 20. Hidden Process Discovery

The output demonstrated that *rubilyncon* process was concealed from the task list. The option *–x [TASKID]* from *volafox* enabled to extract this process for further malware analysis. The built-in system command, *netstat*, did not reveal hidden network session which *volafox* found in **Figure 21**.

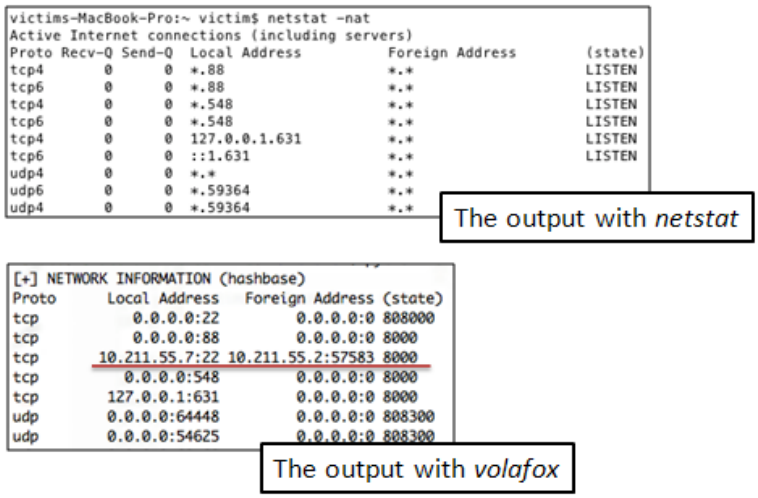

Figure 21. The Output Comparison between *netstat* and *volafox*

5. Conclusion

As the number of Mac OS X exploits has increased, the need of antivirus product for the OS has also boosted. After *Flashback* malware which infected around 600,000 Mac OS, Apple has changed their slogan from "Mac design focus on peace" to "Mac does not infect the PC virus and protect the safety of your data, nothing to do". The *crisis* malware was found at the end of 2012 featured a rootkit for the purpose of concealment as well as user information leakage. The *rubilyn* rookit supported a variety of hiding techniques and released to the public with open source. However, it is true that antivirus for Mac is relatively less effective than that for Windows in that the former has merely adapted signature-based malware detection technique.

One of the efficient methods to detect rootkits is to make use of memory forensics. This technique includes the loaded structure analysis on memory and then data extraction in order to check if there is a rootkit on system. For these reasons, this technique has been widely used to incident response and malware analysis. The memory forensics for Mac OS X has been studied for a couple of years, and several memory forensic tools support Mac as of now.

This paper proposed how to analyze the system infected by Mac OS X rootkit with memory forensics. It covered not only memory forensic methodology in Mac but also rootkit feature description and detection techniques. Furthermore, the case study of *rubilyn* presented how to apply memory analysis tool for rootkit detection with efficiency. If this technique could be put into practice for antivirus product, possibly it might cope with high-level malware attack more effectively.

Acknowledgments

This research was supported by Agency for Defense Development (ADD).

References

[1] New Targeted Attack on Tibetan Activists Using OS X Discovered. The Mac Security Blog, Intego. Feb. 2013.

[2] *rubilyn* rootkit – Mac OS X rootkit, http://seclists.org/fulldisclosure/2012/Oct/55.

[3] OS X ABI File Format Reference, Apple Developer. 2009.

[4] *volafox*: Support New OS!! Mountain Lion xD, n0fate's Forensic Space. http://forensic.n0fate.com/2012/08/volafox-support-new-os-mountain-lion-xd.html.

[5] Jonathan Levin, Mac OS X and iOS Internal : To The Apple Core. Wrox. 2012.

[6] Joseph Kong, Designing BSD Rootkits : An Introduction to Kernel Hacking. No Starch Press. 2007.

[7] I/O Kit Fundamentals, Apple Developer. 2007.

[8] Charlie Miller, Dino Dai Zovi. The Mac Hacker's Handbook. Wiley. 2007.

[9] *volafox*: ToDo, Google Code. http://code.google.com/p/volafox/wiki/ToDo.

[10] TrustedBSD Mandatory Access Control (MAC) Framework. TrustedBSD. http://www.trustedbsd.org/mac.html.

[11] Dionysus Blazakis. The Apple Sandbox. BlackHat DC 2011.

[12] Abusing OS X TrustedBSD Framework to install r00t backdoors.., Reverse Engineering Mac OS X. http://reverse.put.as/2011/09/18/abusing-os-x-trustedbsd-framework-to-install-r00t-backdoors/.

[13] Andrew Case, Mac Memory Analysis with Volatility, DFIR Summit. 2012.

[14] volafox project, Google Code. http://code.google.com/p/volafox.