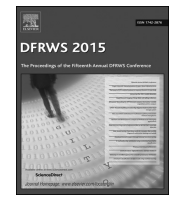
Digital Investigation 14 (2015) S25–S33

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2015 US

Advancing Mac OS X rootkit detection

Andrew Case*, Golden G. Richard III

Volatility Foundation, New Orleans, LA 70001, USA
Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
Memory analysis
Rootkits
Kernel level malware
Digital forensics
Incident response

A B S T R A C T

In the last few years there has been a sharp increase in the use of Mac OS X systems in professional settings. This has led to increased activity in the development of malware and attack toolkits focused specifically on OS X systems, and unfortunately, these increasingly powerful offensive capabilities have not (yet) resulted in better defensive research. Only a few public defensive research efforts currently exist and these only cover a portion of the attack surface that malicious OS X software has access to, particularly regarding kernel-level malware.

In this paper, we present new rootkit detection techniques that attempt to close the gap between offense and defense, with a specific focus on kernel-mode components. The new detection techniques in this paper were motivated by analyzing currently available detection strategies for Windows and Linux, and noting associated deficiencies in detection schemes for Mac OS X. For each missing capability, OS X was studied to see if a similar operating system facility existed and if it could be abused by malware. For those fitting these criteria, new detection techniques were created, and these are discussed in detail in the paper.

For each new rootkit detection technique we propose, a Volatility plugin was developed. Volatility is currently by far the most popular memory forensics framework in incident response and malware analysis, and by incorporating our work into Volatility, it can become immediately useful to the community. The paper concludes with an evaluation of the plugins, to illustrate their usefulness.

Introduction

Historically focused more on Windows and Linux systems, attackers and malware authors have begun turning a critical eye on Mac OS X, due to its increasing use in business, government, non-profit aid groups, and political organizations. While rootkits targeting Windows and Linux have been studied for over a decade, serious interest in OS X has existed for only a few years. As usual, attackers had the initial advantage, as there was little motivation to spend time researching rootkit detection for an operating system that was almost never targeted. This situation has now drastically changed as sophisticated, nation-state backed malware samples have been found that focus extensively on OS X systems and users (Myers, 2013; Katsuki, 2012; Kaspersky, 2014).

These developments have led to increased attention in OS X kernel rootkit detection, with the majority of the public research occurring in late 2013 and throughout 2014. In order to detect existing malware, these research efforts focused almost exclusively on kernel components targeted by previously detected malware samples. This narrow range of detection capabilities leaves ample room for other types of attacks that can fully subvert system security. Previous efforts also ignored several detection capabilities

* Corresponding author.
E-mail addresses: andrew@dfir.org (A. Case), golden@cs.uno.edu (G.G. Richard).

http://dx.doi.org/10.1016/j.diin.2015.05.005

that already exist for Windows and Linux and omitted analysis of many OS X-specific subsystems that we propose are ripe for abuse by malicious code.

In this paper, we discuss several important rootkit detection capabilities for Mac OS X that are absent from current-generation tools. We also discuss the resulting ways in which malware can use these unmonitored operating system facilities to steal data, monitor user activity, and subvert live analysis of the system. These detection gaps were identified by comparing the current anti-kernel rootkit capabilities of Windows and Linux to those of OS X. Since all operating systems provide similar services, we surmised that facilities targeted on Windows and Linux would also exist on OS X. While studying these capabilities, we also discovered OS X-only features that could potentially be abused by malware and that are not covered by current detection techniques.

In order to detect rootkits that may abuse the identified features, we developed Volatility plugins that implement inspection of the relevant subsystems. These new Volatility plugins are presented along with a discussion of their implementation, including associated kernel data structures. The paper concludes with an evaluation of the developed plugins and a discussion of a how similar research effort would likely be beneficial for improving rootkit detection for Windows and Linux systems.

Related work

While memory forensics for OS X systems has received less attention in the research community than for Linux and Windows, there has still been a substantial amount of work. The first major OS X research effort was presented by Matthieu Suiche at Black Hat DC in 2010 (Suiche, 2010). That effort covered the necessary background and data structures for extracting the list of system call table handlers, processes, mounted file systems, and kernel extensions. No source code or tools from this presentation were ever released.

In 2011, the Volafox (Volafox, 2015) project was released. Volafox originated from a fork of Volatility (Volatility, 2015), which at the time only supported Windows and Linux. The Volafox fork involved significant changes to Volatility's internal architecture, making plugins developed for each incompatible with the other. As of this writing, Volafox has support for listing processes along with their file handles, memory mappings, and network connections. It can also recover the list of loaded kernel modules, TrustedBSD policy handlers, and mounted file systems. Due to the design limitations of Volafox, plugins are brittle across kernel versions and adding new plugins requires substantially more work than with the current version of Volatility. The main Volafox developer, Kyeong-Sik Lee, has also published a paper and released a tool that can find Apple KeyChain encryption keys in memory and subsequently open encrypted keychains offline (Lee and Koo, 2012; chainbreaker, 2015).

In 2012, Andrew F. Hay published his master's thesis, which examines the file handling implementation in OS X (Hay, 2011). This leads to recovery of a process's open file handles, understanding mount points, and the first steps in recovering files from the operating system's file system cache. The thesis included Volafox plugins that implemented the described research.

Also in 2012, Volatility implemented full OS X support in the official software release. Although no formal paper was written describing the research, two presentations discuss the initial effort (Case, 2012, 2014). As this paper is written, the latest release of Volatility, version 2.4, has over fifty plugins targeting OS X. Over half of these were added between Volatility 2.3 and 2.4.

In 2013 and 2014, Cem Gurkok submitted several plugins to Volatility during each year's Volatility plugin contest (Gurkok, 2015). These plugins largely focused on detection of OS X kernel rootkits, including DTrace hooks, inline code hooks, and malicious TrustedBSD policy handlers. Many of these plugins have since been integrated into the stable Volatility release. Additional capabilities for Mac OS X analysis continue to be added to Volatility; for example, in 2014, Case and Richard designed Volatility plugins to address Mac OS X's compressed RAM facilities, providing automatic decompression of compressed areas of the physical memory address space (Richard and Case, 2014).

In 2014, the Rekall Memory Forensics framework (Aallievi, 2012) was released. This framework is also a fork of Volatility, and like Volafox, its plugins are incompatible with Volatility. While this framework does have OS X support, it forked an older version of Volatility and, as such, has less than twenty OS X plugins at the time of writing. In particular, it is missing nearly all of the rootkit detection plugins that Volatility developers added in the 2.4 release. As this paper is written, there have been no new OS X rootkit detection plugins added to Rekall that were not derived from Volatility.

Because Volatility is both popular within the incident response and malware analysis communities and currently has the most robust OS X support, particularly for malware detection and analysis, we choose it as our development platform. All plugins described in the paper will be contributed to the Volatility project upon publication of this paper.

New detection capabilities

In this section we present our newly developed OS X rootkit detection capabilities. Each section begins with the motivation behind the detection mechanism, including where possible, real-world rootkits that have targeted this or similar functionality on Mac OS X or other operating systems. We note that several subsystems discussed have not yet been targeted by known Mac OS X malware, but due to the abuse of the equivalent features on Windows and Linux, we investigated them to put the defensive community ahead of malware authors. We then discuss the implementation of the associated kernel subsystem on OS X and how the corresponding Volatility plugin(s) is able to detect malicious use or tampering with the subsystem in question.

Power event notifications

Motivation

Windows provides kernel modules the ability to be notified when shutdown-related events, such as system reboot, shutdown, crash (e.g., blue screen of death), sleep, and hibernation, are about to occur. These notifications are implemented through a callback mechanism, which executes each registered function before allowing the hardware event to proceed. The operating system provides this eventing system so that legitimate kernel components can clean up before the system ceases to operate. Common legitimate uses include freeing resources, flushing disk and network buffers, terminating network connections, and resetting hardware devices to a known state.

On Windows, malware has abused these features for a number of purposes. For instance, Rustock.C registers a crash notification in order to wipe itself from memory before allowing the crash dump to be written (Ligh et al., 2014). Sinowal (Aallievi, 2012) uses these notifications to ensure that its MBR-based persistence mechanism was not removed from disk. TDL3 (Matrosov and Rodionov, 2010) uses shutdown notifications to restore its AutoRun registry keys if anti-virus tools remove them during an attempt to clean the system.

Implementation of event notifications in OS X

The IOKit subsystem provides a generic API to interact with hardware devices and IOServices on the system. Part of this API includes the ability to receive notifications for power-related events. These include sleeping and waking of the system, plus several more, and are defined in the header file *./iokit/IOKit/IOMessage.h* in the Mac OS X kernel source code. The intent to receive notifications is referred to as an "interest" and kernel extensions can call the *registerInterest* function, or one of its higher-level wrappers, to register their notification function.

Internally, interests are registered in the *IOService* I/O Registry plane. The I/O Registry contains several planes, each of which is stored as a tree of *IORegistryEntry* structures. Each node of the tree stores information on a particular component (service, driver, hardware device, etc.). The IOKit planes can be viewed on a live system through the *ioreg* command.

When an interest is registered, it is then added as a property to the *IORegistryEntry* that created it. The properties are stored in the entry's *fPropertyTable* member as a hash table backed by an array of *dictEntry* structures. For interests, the key for each entry is the name of the property ("IOGeneralInterest", "IOAppPowerStateInterest", and so on) and the value is a pointer to an *IOCommand* structure. Each *IOCommand* structure stores a list of registered callbacks for the particular interest in the registry entry. This list's elements are of type *IOServiceInterestNotifier* and this structure's *handler* member is a function pointer to the registered notification routine. When a power event is triggered, all of these functions will be called before the event happens.

Volatility Plugin: mac_interest_handlers

To enumerate all registered power-related event notification handlers, the *mac_interest_handlers* plugin was created. This plugin first finds the root of the IOKit tree, which is stored in the *gRegistryRoot* global variable as an *IORegistryEntry* structure. The tree is then recursively walked using the *fRegistryTable* member. This member stores a hash table whose *IOServiceChildLinks* member can be used to recursively enumerate all child nodes in the *IOService* plane.

For each node in the tree (*IORegistryEntry*), its properties are enumerated and checked for power-related interests. If any are present, each property's value is treated as an *IOCommand* class. The list of handlers tracked by this class are then enumerated and checked to see if they are malicious or not.

For the purposes of our plugins, a handler is deemed malicious if its address is not located within the code section of the running kernel or within the address space of a loaded module in the active modules list. This approach is taken by existing Volatility plugins, and we simply re-used the existing API for this verification. We note if malicious modification is made to the code of the running kernel, then the existing *mac_apihooks* plugin will detect this tampering.

Kernel timers in Mac OS X

Motivation

Timers allow kernel components to set timers and register functions to be executed when the timers expire. These facilities are present in all major operating systems and support correct operation of kernel components such as deadlock watchdogs, network congestion monitoring, queue clearing, and handling of file system buffering. Timers have also been used by many different malware families to periodically flush buffers of logged keystrokes to disk, contact command and control servers, and check that persistence mechanisms have not been removed. While a timer itself is not necessarily malicious, its presence can point an analyst to a previously unknown kernel driver or code region, and the timer function can then be analyzed to determine its purpose.

OS X implementation of kernel timers

OS X kernel extensions can register timers that specify a period of time to elapse along with a function to be called when the timer expires. These timers are stored in the *rtclock_timer* member of each processor's CPU-specific variables. This member is of type *rtclock_timer* and stores a queue of timers assigned to the CPU. Each element of this queue is stored as a *call_entry* structure, which tracks when the timer is to elapse, the function to be called, and the address of two optional parameters to the function. These parameters can be set when the timer is created.

Volatility plugin: mac_timers

The *mac_timers* plugin was created to enumerate the registered kernel timers within a memory sample. This plugin first determines the number of active processors by using the *real_cpus* kernel variable, and then enumerates

data associated with each active processor by using the *cpu_data_ptr* array. This array holds one *cpu_data* structure per processor.

For each processor, its *rtclock_timer* member is examined and its queue of timers enumerated. For each timer, the time to elapse, address of each parameter, and address of the handler function is printed along with an indication of whether the handler is deemed to be suspicious. This will effectively detect any malicious kernel components that have registered timers.

Driver communication in Mac OS X: devfs

Motivation

When a driver needs to implement communication with its userland components, it must set up handlers for the range of operations that processes can perform. These include opening a reference to the driver, read and write operations, memory mapping, close operations, and more. Individual drivers register a set of function pointers that the kernel calls when a userland component makes a request of the driver.

These handler tables have been of interest to malware researchers for two main reasons. The first is that malware will often overwrite the function pointers of targeted drivers in order to insert and hide data. For example, by hooking the *read* operation of a driver, malware can filter all data returned to userland. This can include information such as file contents, directory listings, or network packet bytes. By hooking the *write* function of a file system driver, rootkits can protect their own files from modification or deletion by preventing a security tool's write operations from impacting the contents of a device. Rootkits, such as TDL3, implement this write-filtering technique and numerous rootkits have implemented read filtering.

The second reason the handler tables are of interest is that malicious drivers will set up their own handlers for userland components of a malware kit to communicate with. These drivers often provide functionality such as process, file, and network connection hiding, privilege escalation, and hiding of logged in users. By enumerating such handlers, malware researchers can very quickly determine a significant portion of the malware's capabilities.

OS X implementation of devfs

OS X provides two methods for drivers to communicate with their userland components. The first, described in this section, is through *devfs*. The second, described in Section 3.4, is via IOKit.

devfs is defined by POSIX, and as such, exists on both Linux and OS X. It supports the creation and handling of userland accessible files under the */dev* directory. Every character and block device on the system, such as a keyboard, mouse, hard drive, sound card, etc. will have a node (file) under *devfs* that userland tools can access via standard filesystem functions, such as *open*, *read*, *write*, and *mmap*, to interact with the device.

Kernel extensions that wish to create a character device call the *cdevsw_add* function with a pre-populated *cdevsw* structure. This structure has a function pointer for every operation that can be requested of the device. To create a block device, the *bdevsw_add* function is called with a pre-populated *bdevsw* structure. Similar to *cdevsw*, this structure has function pointers for block device operations, such as *open*, *close*, and *ioctl*.

Volatility plugin: mac_devfs

The *mac_devfs* plugin was created to enumerate all character and block devices and validate their handler tables. To find all of the devices, Volatility's existing capability to examine file system information in a memory dump is used, allowing our plugin to programmatically enumerate files in the */dev* branch of the file system. The type of each file in this directory and its sub-directories is checked using the *v_type* member of each file's *vnode* structure. A character device has type “VCHR” while block devices have type “VBLK”.

The *v_data* member of *vnode* holds a pointer to file type specific structures. For character and block devices, it holds a pointer to the device's *devnode* structure. This structure holds metadata about the file, such as its MAC times and owner information, as well as its major and minor number. To retrieve a device's set of userland handling operations, its major number must be indexed into the global *cdevsw* array. This array holds elements of type *cdevsw*, which in turns holds all of the operation pointers. Our plugin validates these pointers and reports suspicious (potentially hijacked) operations.

Driver communication in Mac OS X: IOKit

Motivation

As discussed in the previous section, OS X provides two methods for drivers to allow userland components to send requests. This section describes the IOKit method, the motivation for which is the same as the previous.

OS X IOKit implementation

IOKit (IOKit, 2015) allows userland clients to dynamically find devices of interest based on their properties, in contrast to *devfs*, which relies on clients having knowledge of existing device names (e.g., */dev/sda* or */dev/mem*). Once a device is found, it can be opened with the *IOServiceOpen* function. This returns a handle similar to the POSIX *open* function. This handle can then be passed to the *IOConnectMethod* family of functions in order to send and receive data.

The implementation for this method of communication is a mix of C++ member functions and developer-defined external functions. The C++ member functions are setup and teardown functions that each device must implement. The developer-defined functionality is implemented in the *getTargetAndMethodForIndex* function of the class. This function receives an integral index that defines the functionality to be performed. The driver and its userland components must use a predetermined protocol in order to map indexes to functionality.

Volatility plugin: mac_kernel_classes

After a deep investigation of IOKit internals on Mac OS X, we developed a new Volatility plugin,

mac_kernel_classes, which detects tampering with IOKit. In particular, the plugin finds all loaded IOKit C++ classes and then verifies their virtual tables (*vtables*). To start, the plugin locates the *sAllClassesDict* global variable. This variable points to a hash table that tracks all C++ classes loaded by kernel components and kernel extensions.

The keys of the hash table are the names of the loaded classes and their associated values are pointers to the *MetaClass* member of each class. *MetaClass* holds a pointer to the virtual table used by every instance of the class type, which is stored as a NULL-terminated array of function pointers.

Our plugin enumerates every loaded class, walks each *vtable*, and reports whether each handling function is within a known kernel component. This effectively verifies all virtual functions within every loaded class. While we are not aware of any malicious hijacking of *vtable* operations in IOKit on OS X, we are aware of benevolent uses of virtual table overwriting (e.g., in (Brocker and Checkoway, 2014), to prevent tampering with webcam LED indicators). Furthermore, exploits against every major platform have targeted C++ *vtable* entries in one form or another, as they allow for trivial code execution during the exploitation phase. It is therefore prudent to pay attention to potential virtual table exploitation in the Mac OS X kernel.

The user-defined functions related to *getTargetAndMethodForIndex* are also of potential forensic interest, but due to the flexible nature in which the client and driver communicate, we were unable to develop a generic method to determine if functions have been hooked or not and a solution to this problem remains an open question.

File system hooking in Mac OS X

Motivation

Much like *devfs* devices, file system implementations on Linux and OS X utilize sets of function pointers that correspond to specific file operations. These isolate core kernel functionality from the specific implementation of particular filesystems, and include support for reading from and writing to a file, getting the list of files in a directory, and querying metadata associated with a file. Also like *devfs* devices, hooking of these pointers can lead to filtering of results that are obtained by code that interacts with the file system.

Particularly on Linux, file system hooking has been abused by rootkits since the early 2000s. A commonly seen technique on this platform is hooking of the read directory function of the *procfs (/proc)* file system. Under Linux, processes are enumerated by reading each per-process directory stored under */proc*. By filtering directories returned by this operation, a kernel mode rootkit can effectively hide processes from tools such as *ps* and *top*. Similarly, hooking of the read function of */proc/net/tcp* can be used to hide network connections from *netstat* and *lsof*.

Mac OS X VFS implementation

OS X implements nearly the same virtual file system (VFS) API as Linux. For VFS, arrays of function pointers specific to particular filesystems handle I/O operations, isolating the kernel from the specifics of a particular filesystem. Generic system calls such as *read* operations ultimately call a filesystem-specific *read* operation via the function pointer array associated with a mounted filesystem. The array of pointers is a popular attack target, since it allows filesystem operations to be easily subverted. Since OS X provides the same functionality in the same form, it is also vulnerable to the same attacks as other systems that implement a VFS, notably Linux.

During our research we located two places where VFS hooks could be placed by malicious kernel extensions in order to filter content. The first was in the configuration table for a particular file system. This structure, of type *vfstable*, stores function pointers for getting and setting file system attributes, syncing the file system cache to disk, and mapping VFS internal identifiers to actual files in the file system. These pointers are stored in the *vfc_vfsops* member, which is of type *vfsops*.

The second filtering method found was hooking of the *vnode* operations structure of each file system. The set of operations are stored as a *vnodeopv_entry_desc* structure, whose *opve_impl* member stores a pointer to the handler routine and whose *vnodeop_desc* member stores information such as the routine's name (e.g. “vnop_open”, “vnop_read”, “vnop_write”). This structure represents every operation that code running in the kernel or userland can request from the file system.

Hooking of the functions described in the two previous methods can facilitate file and directory hiding, false reporting of file system metadata and content, prevention of file modification, and much more, all of interest to rootkit developers.

Volatility plugin: mac_check_fop

The *mac_check_fop* plugin was developed to detect these types of rootkits. Its implementation was inspired by the *linux_check_fop* plugin. *mac_check_fop* currently performs two sets of checks. The first check walks all *vfstable* structures, accessible via an array in the global *vfstbllist* kernel variable, and determines whether each operation's function pointer is suspicious.

The second check enumerates each *vnode* operations structure by locating the *vfs_opv_descs* global variable. This stores a *vnodeopv_desc* structure for each loaded file system. The *opv_desc_ops* member of *vnodeopv_desc* is of type *vnodeopv_entry_desc*, whose members were explained previously. Once the entry descriptor is obtained, its handler structure is then checked for any malicious entries.

File system events

Motivation

File system event monitoring allows code both in the kernel and in userland to monitor for file events, such as creation, deletion, modification, renaming, and others. This functionality can be used by rootkits for a wide variety of purposes including monitoring its own files for modification or removal by security tools, monitoring files it wants to steal data from, and monitoring for installation of security and forensics tools. Since this monitoring must be implemented within the kernel, even if a userland process

requests the monitor, kernel data structures will perform the tracking of event handlers. This has the interesting advantage of allowing kernel memory analysis to pinpoint suspicious processes on a system.

File system monitoring in OS X

As demonstrated in FSLogger (fslogger, 2015), OS X provides robust capabilities for userland tools to monitor file activity by interacting with the */dev/fsevents* character device. In particular, userland tools can monitor for file creation, deletion, content and attribute modification, renaming, and more. Each registered event "watcher", namely a process that has registered for events, is stored within the *watcher_table* global array. The array is capped at a maximum of eight watchers. Each watcher is represented by an *fs_event_watcher* structure, which tracks the process name (*proc_name*) and PID (*pid*) that registered the watcher along with the specific events it is interested in. The list of events is stored in the *event_list* member as a byte mask.

Volatility plugin: mac_vfsevents

The *mac_vfsevents* plugin was developed in order to enumerate watchers. The plugin first finds the global table of watchers and then enumerates it. For each watcher found, the process information is printed along with the list of watched events.

Since userland components interact with file system events by opening and then reading from */dev/fsevents*, there is no function pointer that can be used to determine where in the process' memory the events are handled. Instead, existing Volatility plugins, such as *mac_procdump* and *mac_librarydump*, must be used to extract the executable of interest to disk in order to support further reverse engineering.

Eventing in Mac OS X

Motivation

While researching the other components discussed in this paper, the authors also investigated internals of the *kqueue* interface provided by OS X (Lemon, 2015), derived from the facility of the same name in FreeBSD (dating to FreeBSD 4.1). This interface provides userland components the ability to monitor file events, file descriptor events, process events, including creation, and signals sent to a process. With sufficient privileges these can be abused for a large range of possibilities including preventing processes from loading, detecting when files and network connections are being tampered with, and reacting after a process of interest has terminated. As with file system events, by analyzing the *kqueue* facility in kernel memory, a forensics tool can detect suspicious processes that should be marked for further investigation.

OS X KQueue implementation

Userland tools can use *kqueue* and its companion *kevent* call to monitor a wide range of activity on the system. Examples of supported file-descriptor based events include when a socket is being read from, when a new connection is being established, and when a file is being deleted, written to, or renamed. Processes can be monitored for termination, forking, and calling *execve* by supplying the PID of the process to monitor.

Each registered *kevent* handler is represented by a *knote* structure, which is linked to its related process in several ways. File related *knote* structures are stored within the *p_fd* member of the process' *proc* structure. This member is of type *filedesc* and its *fd_knlist* holds a list of *knote*s while its *fd_knhash* member stores a hash table of members. *knote*s for tracking process events are stored as a list in the *p_klist* member of struct *proc*. *knote*s for sockets are stored within the send or receive queue member of the socket or within the socket's *so_klist* member. For completeness, all three lists must be enumerated to gather all of a socket's *knote*s.

The *knote* structure itself holds several pieces of information including linkage with the other *knote*s, a pointer to its filtering operation structure, and information about the event being monitored. The monitoring information is stored in the *kn_kevent* member, which is of type *kevent64_s*. Decoding the event requires analysis of its *ident* and *filter* members. The *filter* member specifies the type of monitor (e.g., *EVFILT_READ* for monitoring file descriptor reads or *EVFILT_PROC* for monitoring process events). The *ident* member's value depends on the type of event being filtered. For process events it will be the PID to monitor and for file or socket events it will be the file descriptor of the opened file or active network connection to monitor.

Volatility plugin: mac_kevents

The *mac_kevents* plugin recovers all registered file, socket, and process *knote*s across all active processes. This is accomplished in several steps. To start, each process descriptor is enumerated using existing Volatility APIs. For each process found, its *p_klist* member is walked to recover registered process filters. After this, the file descriptor filters are walked by following the *p_fd* member, as described previously. To recover the socket filters, each file descriptor in the process' file descriptor table is checked and for files of type *DTYPE_SOCKET*, the *socket* structure is located and the general, send queue, and receive queue filters enumerated for analysis.

As the *knote*s are discovered, their event structures are decoded and output in an investigator-friendly manner. To decode events, the *ident* and *filter* members are parsed as described previously, along with the *fflags* member, as necessary. For file-related filters (*EVFILT_VNODE*), *fflags* specifies the event to monitor (delete, write, rename, etc.), and for process filters *fflags* specifies the process events to monitor (fork, exit, exec, etc.).

Evaluation

Overview

The plugins developed during this research were tested on OS X versions 10.6 (Snow Leopard) through 10.9 (Mavericks). Volatility did not have official support for 10.10 (Yosemite) at the time of writing so we could not perform conclusive testing. From our reading of the Yosemite kernel source code it appears as if all the facilities targeted during

this research work in essentially the same way and the plugins will be updated as needed as the official Volatility release catches up.

Testing was performed on publicly available OS X memory samples (AMF, 2015; PAMS, 2015) as well as samples generated during the research. Memory images on native hardware were generated with Mac Memory Reader and virtual machine guests' memory was captured using snapshot support in VMware Fusion. Since the time we conducted this research, Mac Memory Reader is no longer available for download, so an alternative must be used to acquire memory on native hardware.

Expected data to compare with plugin output was generated through study of the kernel source code. Cross-references to the functions that fill the analyzed data structures were used to determine what types of data was being populated and the exact values. These were then compared to the information recovered by the plugins. For several of the plugins, closed-source Apple kernel extensions interacted with the subsystems. A subset of these were verified through binary analysis of the extensions.

Plugin output and usability

In this section we demonstrate several of the discussed plugins and show how they can be useful to a forensic or malware analyst during an investigation.

mac_devfs against crisis

Crisis (Katsuki, 2012) is arguably the most infamous and powerful malware to ever be discovered to impact OS X systems. Its primary purpose is espionage and it contains both userland and kernel components that can record audio and activate web cameras, take pictures, steal login credentials, perform keystroke logging, and more.

In order for the userland components to be able to request functionality from the kernel components, Crisis creates a character device named */dev/pfCPU* (Vilaca, 2012; Nayyar, 2014). The userland component can then use *ioctl* requests to send commands and receive replies.

When executed against infected memory samples, our *mac_devfs* plugin locates this character device as well as information about the handlers for all its operations. The plugin also identifies the kernel component that implements the handler, and if possible, the name of the symbol within the executable. Fig. 1 illustrates the output of the plugin against, filtered to concentrate only on the Crisis device.

As Fig. 1 illustrates, Crisis implements *d_ioctl*, *d_open*, and *d_close* functions for the */dev/pfCPU* device. Other handler functions simply point to the generic handler for unsupported operations. When Crisis' userland components perform an *ioctl* operation on the */dev/pfCPU* device, the function associated with the *d_ioctl* function pointer will be executed within the kernel. Since our plugin provides the function pointer address (*0xffffff7f808049c6* in this case), the analyst can then immediately perform binary analysis on the function in order to determine its purpose. Since the name of the module is listed, it can also be extracted in its entirety through the existing *mac_moddump* plugin. This allows for additional, deeper analysis using static analysis tools like IDA Pro.

We note that Crisis does attempt to hide its kernel module, but Volatility's existing module discovery algorithm is not fooled by the attempt, and as such the module's name is reported. If a handler pointed to an area that Volatility could not map back to the kernel itself or a kernel extension, the module would be listed as UNKNOWN. This is a common Volatility convention to point out suspicious regions within memory.

mac_vfsevents against FSLogger

FSLogger, described previously, is an open source tool for filtering and displaying all file system events on the system. While its primary purpose is not malicious, its behavior is representative of malware that monitors system events and

$ python vol.py --profile=MacLion_10_7_3_AMDx64 -f crisis-infected.snap mac_devfs

Offset	Path	Member	Handler	Module	Handler Sym
0xffffff8000859000	/dev/pfCPU	d_mmap	0xffffff800055e480	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_ioctl	0xffffff7f808049c6	com.apple.mdworker	
0xffffff8000859000	/dev/pfCPU	d_strategy	0xffffff800055e490	__kernel__	_enodev_strat
0xffffff8000859000	/dev/pfCPU	d_select	0xffffff800055e480	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_read	0xffffff800055e480	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_write	0xffffff800055e480	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_reserved_1	0xffffff800055e48	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_open	0xffffff7f808049b6	com.apple.mdworker	
0xffffff8000859000	/dev/pfCPU	d_close	0xffffff7f808049be	com.apple.mdworker	
0xffffff8000859000	/dev/pfCPU	d_reset	0xffffff800055e480	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_reserved_2	0xffffff800055e480	__kernel__	_enodev
0xffffff8000859000	/dev/pfCPU	d_stop	0xffffff800055e480	__kernel__	_enodev

Fig. 1. *mac_devfs* plugin output against a 10.7.3 memory image of machine infected with Crisis.

$ python vol.py --profile=MacMavericks_10_9_5_AMDx64 -f fslogger.dump mac_vfsevents

Offset	Name	PID	Events
0xffffff80df4b6008	coreservicesd	36	DELETE, RENAME
0xffffff80df4e2008	fseventsd	39	CREATE_FILE, DELETE, STAT_CHANGED, RENAME, CONTENT_MODIFIED, EXCHANGE, FINDER_INFO_CHANGED, CREATE_DIR, CHOWN, XATTR_MODIFIED, XATTR_REMOVED
0xffffff80df69e008	mds	63	CREATE_FILE, DELETE, STAT_CHANGED, RENAME, CONTENT_MODIFIED, EXCHANGE, FINDER_INFO_CHANGED, CREATE_DIR, CHOWN, XATTR_MODIFIED, XATTR_REMOVED
0xffffff80dfc21008	fslogger	8495	CREATE_FILE, DELETE, STAT_CHANGED, RENAME, CONTENT_MODIFIED, EXCHANGE, FINDER_INFO_CHANGED, CREATE_DIR, CHOWN, XATTR_MODIFIED, XATTR_REMOVED

Fig. 2. *mac_vfsevents* plugin output with FSLogger executing for a 10.9.5 memory image.

so it makes a good test case for our *mac_vfsevents* plugin. Fig. 2 illustrates the output of *mac_vfsevents* with the *FSLogger* tool active.

As can be seen, there are four processes listed along with the events they are monitoring. The *coreservicesd*, *fseventsd*, and *mds* processes are all legitimate OS X components that always register *vfsevents* monitors. This makes sense as the core services daemon provides a wide array of functionality, including file system monitoring, *fseventd* runs the file events subsystem in userland, and *mds* is a component of Spotlight, which indexes all files upon modification, among other indexing operations.

The fourth entry, *fslogger*, is our version of *FSLogger* set to monitor all possible events. This output line would immediately indicate to an analyst that there is an process on the system monitoring all events and that further investigation of the process' purpose and capabilities is needed.

mac_check_fop against a clean sample

Fig. 3 shows select output from the *mac_check_fop* plugin against a clean 10.7.3 64 bit sample. The output shows a number of operations (*lookup*, *create*, *open*, *close*, *rename*, *readdir*, *mkdir*, etc.) for the HFS + file system. Every

$ python vol.py --profile=MacLion_10_7_3_AMDx64 -f clean.sample mac_check_fop

Offset	Name	Handler	Module	Handler Sym
0xffffff800083f710	_hfs_vnodeop_opv_desc/vnop_lookup	0xffffff80004e3610	__kernel__	_hfs_vnop_lookup
0xffffff800083f720	_hfs_vnodeop_opv_desc/vnop_create	0xffffff80004fc700	__kernel__	_hfs_vnop_create
0xffffff800083f730	_hfs_vnodeop_opv_desc/vnop_mknod	0xffffff80004fc110	__kernel__	_hfs_vnop_mknod
0xffffff800083f740	_hfs_vnodeop_opv_desc/vnop_open	0xffffff8000501680	__kernel__	_hfs_vnop_open
0xffffff800083f750	_hfs_vnodeop_opv_desc/vnop_close	0xffffff80004ffcd0	__kernel__	_hfs_vnop_close
0xffffff800083f760	_hfs_vnodeop_opv_desc/vnop_getattr	0xffffff8000501b00	__kernel__	_hfs_vnop_getattr
0xffffff800083f770	_hfs_vnodeop_opv_desc/vnop_setattr	0xffffff8000501160	__kernel__	_hfs_vnop_setattr
0xffffff800083f780	_hfs_vnodeop_opv_desc/vnop_read	0xffffff80004eaa20	__kernel__	_hfs_vnop_read
0xffffff800083f790	_hfs_vnodeop_opv_desc/vnop_write	0xffffff80004ea080	__kernel__	_hfs_vnop_write
0xffffff800083f7a0	_hfs_vnodeop_opv_desc/vnop_ioctl	0xffffff80004e8fb0	__kernel__	_hfs_vnop_ioctl
0xffffff800083f7e0	_hfs_vnodeop_opv_desc/vnop_mmap	0xffffff80005008e0	__kernel__	_hfs_vnop_mmap
0xffffff800083f7f0	_hfs_vnodeop_opv_desc/vnop_fsync	0xffffff80004ff3d0	__kernel__	_hfs_vnop_fsync
0xffffff800083f800	_hfs_vnodeop_opv_desc/vnop_remove	0xffffff80004fe800	__kernel__	_hfs_vnop_remove
0xffffff800083f810	_hfs_vnodeop_opv_desc/vnop_link	0xffffff80004e2f70	__kernel__	_hfs_vnop_link
0xffffff800083f820	_hfs_vnodeop_opv_desc/vnop_rename	0xffffff80004fd850	__kernel__	_hfs_vnop_rename
0xffffff800083f830	_hfs_vnodeop_opv_desc/vnop_mkdir	0xffffff80004fc0e0	__kernel__	_hfs_vnop_mkdir
0xffffff800083f840	_hfs_vnodeop_opv_desc/vnop_rmdir	0xffffff80004feb60	__kernel__	_hfs_vnop_rmdir
0xffffff800083f850	_hfs_vnodeop_opv_desc/vnop_symlink	0xffffff80004fd220	__kernel__	_hfs_vnop_symlink
0xffffff800083f860	_hfs_vnodeop_opv_desc/vnop_readdir	0xffffff80005002b0	__kernel__	_hfs_vnop_readdir

Fig. 3. *mac_check_fop* plugin output against a clean 10.7.3 memory image.

HFS + file that is opened from a process or the kernel and then interacted with will have the requested operations routed through these pointers. As with *mac_devfs*, the handlers are checked to be certain they are within a valid memory range. Since these pointers are not hooked, they are printed along with the component that implements them, the kernel, as well as the exact symbol (function name) inside the kernel executable.

Conclusions and future work

This paper has demonstrated that current rootkit detection techniques for OS X systems, particularly for rootkits that target kernel data, are inadequate. We discussed several Mac OS X subsystems that could be abused by rootkits and for which the abuse would not be detected by currently available tools and techniques. These included portions of the IOKit Framework, VFS event monitors, *kqueue* monitors, timers, and more. In order to advance the state of the art in OS X rootkit detection, these gaps in available malware detection were filled through a detailed study of the relevant kernel subsystems, along with development of Volatility plugins that can automate the processing.

For each plugin we developed, we extracted actionable data that the investigator could then focus on to better understand the associated malicious behavior. This included the exact address where malicious code was stored, and, where possible, the name of the kernel component and symbol implementing the malicious functionality. We also discussed how our plugins compliment existing Volatility features, such as automated extraction of process executables and kernel extensions, in order to better frame the output of our plugins in a more detailed reverse engineering or antivirus scanning effort. All of the plugins described in this paper will be freely available after the publication of the paper.

In our view, the work presented in this paper has considerably improved the state of the art for Mac OS X rootkit detection. We are confident that a similar comparison of Windows and Linux detection capabilities will reveal gaps in current generation rootkit detection schemes for each operating system. Results from such a comparison would then require similar research to address the gaps. As an example, we note that while Linux also has *devfs* and *ioctl* facilities, currently no tools are able to list the operation handlers of enumerated *devfs* nodes. As with Crisis and */dev/pfCPU*, malware on both Linux and Windows systems use *ioctl* facilities for communication with userland components. Furthermore, both Windows (MDMS, 2015) and Linux (inotify, 2015) provide file system monitoring from userland, but no existing forensic tool is able to locate and list registered handlers. Finally, subsystems on Linux, such as *dbus* (dbus, 2015), allow registering interest in hardware events, such as USB insertion. Similar subsystems have been abused by malware on Windows, including by Stuxnet (Ligh, 2011), to infect USB devices on insertion.

Acknowlegments

We would like to thank Alex Radocea, Pedro Vilaça, Sarah Edwards, AAron Walters, and the anonymous reviewers for their contributions, which substantially improved the initial draft of the paper. We also acknowledge the generous support of the National Science Foundation (9534); this work was supported in part by NSF Award # 1409534.

References

Aallievi A. Sinowal: MBR rootkit never dies!. 2012. news.saferbytes.it/analisi/2012/06/sinowal-mbr-rootkit-never-dies-and-it-always-brings-some-new-clever-features/. SaferBytes.

AMF. Art of memory forensics supplementary materials. 2015. www.memoryanalysis.net/#!amf/cmg5.

Brocker M, Checkoway S. iSeeYou: disabling the MacBook webcam indicator led. In: Proceedings of the 23rd USENIX conference on Security Symposium. USENIX Association; 2014. p. 337–52.

Case A. Mac memory analysis with Volatility. In: 2012 SANS DFIR Summit; 2012.

Case A. Hunting mac malware with memory forensics. In: 2014 RSA USA conference; 2014.

chainbreaker. https://github.com/n0fate/chainbreaker; 2015.

dbus. dbus documentation. 2015. www.freedesktop.org/wiki/Software/dbus/.

fslogger. osxbook.com/software/fslogger/download/fslogger.c; 2015.

Gurkok C. What's in your silicon?. 2015. siliconblade.blogspot.com/. Silicon Blade Blog.

Hay A. Forensic memory analysis for Apple OS X (Master's thesis). Air Force University; 2011.

inotify. inotify linux man pages. 2015. linux.die.net/man/7/inotify.

IOKit. Introduction to I/O kit fundamentals. 2015. developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/Introduction.html.

Kaspersky. Kaspersky lab uncovers “the mask”. 2014. www.kaspersky.com/about/news/virus/2014/Kaspersky-Lab-Uncovers-The-Mask-One-of-the-Most-Advanced-Global-Cyber-espionage-Operations-to-Date-Due-to-the-Complexity-of-the-Toolset-Used-by-the-Attackers.

Katsuki T. Crisis: the advanced malware. 2012. www.symantec.com/connect/blogs/crisis-advanced-malware.

Lee K, Koo H. Keychain analysis with Mac OS X memory forensics. 2012. forensic.n0fate.com/wp-content/uploads/2012/12/Keychain-Analysis-with-Mac-OS-X-Memory-Forensics.pdf.

Lemon J. KQUEUE(2), FreeBSD man pages. 2015. www.freebsd.org/cgi/man.cgi?query=kqueue&;sektion=2.

Ligh MH. Stuxnet's footprint in memory with Volatility 2.0. In: MNIN security Blog; 2011. mnin.blogspot.com/2011/06/examining-stuxnets-footprint-in-memory.html.

Ligh MH, Case A, Levy J, Walters A. The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory. Wiley; 2014.

Matrosov A, Rodionov E. TDL3: the rootkit of all evil?. In: ESET; 2010.

MDMS. Microsoft directory management functions. 2015. msdn.microsoft.com/en-us/library/windows/desktop/aa363950(v=vs.85).aspx.

Myers L. New OS X malware: another Tibet variant found. 2013. www.intego.com/mac-security-blog/os-x-malware-tibet-variant-found/.

Nayyar H. An opportunity in Crisis. SANS Institute InfoSec Reading Room; 2014. www.sans.org/reading-room/whitepapers/threats/opportunity-crisis-34600.

PAMS. Publicly available memory samples. 2015. github.com/volatilityfoundation/volatility/wiki/Memory-Samples.

Richard GG, Case A. In lieu of swap: analyzing compressed RAM in Mac OS X and Linux. Digit Investig 2014;11:S3–12.

Suiche M. Advanced Mac OS X physical memory analysis. In: Blackhat DC security conference; 2010.

Vilaca P. Tales from Crisis, chapter 3: the Italian rootkit job. 2012. reverse.put.as/2012/08/21/tales-from-crisis-chapter-3-the-italian-rootkit-job/. Reverse Engineering Mac OS X blog.

Volafox. Volafox memory analysis framework. 2015. https://code.google.com/p/volafox/.

Volatility. Volatility memory analysis framework. 2015. github.com/volatilityfoundation/volatility.