Spring 5-13-2016

Detecting Objective-C Malware through Memory Forensics

Andrew Case

University of New Orleans, New Orleans, acase@uno.edu

Detecting Objective-C Malware through Memory Forensics

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
In
Computer Science
Information Assurance

by

Andrew Case

B.S. University of New Orleans, 2009

May 2016

Table of Contents

List of Figures

Abstract

Memory forensics is increasingly used to detect and analyze sophisticated malware. In the last decade, major advances in memory forensics have made analysis of kernel-level malware straightforward. Kernel-level malware has been favored by attackers because it essentially provides complete control over a machine. This has changed recently as operating systems vendors now routinely enforce driving signing and strategies for protecting kernel data, such as Patch Guard, have made userland attacks much more attractive to malware authors.

In this thesis, new techniques for detecting userland malware written in Objective-C on Mac OS X are presented. As the thesis illustrates, Objective-C provides a rich set of APIs that malware uses to manipulate and steal data and to perform other malicious activities. The novel memory forensics techniques presented in this thesis deeply examine the state of the Objective-C runtime, identifying a number of suspicious activities, from keystroke logging to pointer swizzling.

Keywords: Memory Forensics, Objective C, Malware Detection, Volatility, Incident Response

1. Introduction

1.1. Memory Forensics

Memory forensics has quickly become one of the primary methods for digital forensic investigators to detect and analyze sophisticated malware and rootkits. This method of analysis operates by reproducing the algorithms and data structures of an operating system and its applications in an offline manner. By not relying on the live system and its APIs to determine system state, memory forensics tools can acquire artifacts not available to normal system programmers as well as from APIs that malware has manipulated.

1.2. Recent Operating System Hardening Strategies

To date, the bulk of memory forensics research has targeted kernel level analysis. This occurred because kernel-level rootkits wield great power over running systems, including control of hardware devices, the operating system itself, as well as all running applications. Kernel level rootkits also make it trivial for attackers to hide a wide range of activity, such as installation of attacker tools, lateral movement, and long-term, persistent infection.

This model for malware has recently changed as operating systems have heavily locked down access to kernel mode by unknown third party code and taken steps to attempt to protect kernel-level data structures and code from manipulation. The most prominent examples of this trend are the enforcement of signed drivers on Microsoft Windows [1] and Mac OS X [2] as well as Microsoft Patch Guard [3]. While all of these protections have been temporarily bypassed, the discovered vulnerabilities were subsequently patched. Regardless, the protections still significantly raise the bar for attackers to successfully load their rootkits on compromised systems [4, 5, 6, 7].

1.3. Userland Malware

The inability to utilize kernel-level malware has led to a rise in malware that operates mostly in process memory, also known as userland. This malware can accomplish many of the same tasks as kernel-level malware, such as hiding attacker activity from live system tools, stealing data, and maintaining long-term persistence, without having to enter kernel mode. On Windows, this has led to malware with a single executable that that can run on a wide variety of platforms, from Windows XP through Windows 8 and 10. Such broad OS support would be very difficult to do in a stable manner for any kernel-level rootkit with complex functionality. On Mac OS X, this has led to high-profile malware, such as Ventir [8] and Crisis [9], which contain both userland and kernel mode components that load separately depending on whether they are executed with root privileges. Due to the extensive APIs provided by OS X, these malware samples can accomplish the same goals regardless of which components load.

1.4. Objective-C

The novel contributions of this thesis target Objective-C, a language and associated runtime supported by Apple for development of userland applications on the OS X and iOS platforms. As discussed throughout the thesis, malware can abuse the rich APIs of the Objective-C runtime system in order to monitor, steal, and manipulate a wide range of data processed by applications. Unfortunately, these abuses are completely ignored by existing memory forensics research and tools. In order to detect malware using these facilities, research was performed in order to produce new memory forensics analysis techniques that can deeply examine the state of the Objective-C runtime inside of targeted processes. These new defensive techniques were developed against the open source Volatility Memory Analysis Framework [32]. Volatility is one of the most popular memory forensics platforms and is considered an industry standard tool in the fields of incident response and malware analysis. In Chapter 4, each of the developed techniques is presented along with a newly created Volatility plugin that implements the described analysis. Upon publication of this thesis, the plugins will be contributed to the open source Volatility project for use by the forensics community.

1.5. Organization

This remainder of this thesis is broken down into five sections. Section 2 discusses previous research related to this research effort. Section 3 discusses Objective-C and its components that are relevant to memory forensic analysis. Section 4 discusses common methods that malware employs to abuse Objective-C. It also discusses the novel techniques developed for this thesis that allow for detection of such malware. Section 5 provides our roadmap of future work, and section 6 provides references to all external sources discussed in this thesis.

2. Related Work

Although no previous memory forensic analysis efforts exist for deep analysis of Objective-C applications, there has been substantial work in a number of related areas. These efforts are discussed in this chapter.

2.1. Open Source Frameworks

The first open source memory forensics framework to support OS X memory analysis was the Volafox [39] project. Volafox is a plugin-incompatible fork of Volatility. At the time of Volafox's creation, Volatility had no OS X support. For the older operating system versions that Volafox supports, it provides plugins for listing processes as well as each processes file handles, memory mappings, and network connections. It can also recover kernel information, such as TrustedBSD policy handlers, loaded kernel modules, and mounted file systems. Unfortunately, Volafox analysis is very sensitive to particular kernel versions and requires substantial developer effort to make plugins portable across operating system versions.

Volatility, which is the most widely used memory forensics framework in digital forensics and incident response, gained OS X support in 2012. This work was described by two presentations [40, 41], and the code was incorporated into the Volatility 2.3 release. As of version 2.5, Volatility has over 60 plugins for OS X analysis. Due to its popularity and widespread use throughout the forensics community, Volatility was chosen as the development framework for this thesis.

In 2014, the Rekall memory forensics framework added OS X support. Rekall is a fork of Volatility like Volafox, and also like Volafox, Rekall analysis plugins are incompatible with Volatility. As of writing, Rekall has less than twenty OS X plugins, many of which are simply rewrites of the initial set of Volatility OS X plugins.

2.2. Research Efforts

The first public effort of OS X memory analysis was performed by Matthieu Suiche and presented at Black Hat DC in 2010 [38]. This research covered the data structures and algorithms necessary to reconstruct mounted file systems, kernel extensions, active processes, and system call entries. While his paper described his research effort, no related source code was ever released.

In 2012, Andrew F. Hay published his master's thesis, which examined the file manager subsystem of OS X [42]. His research documented how to map opened files to processes, how to determine metadata of opened files, and the first steps towards recovering file contents from memory. His work was incorporated into the Volafox memory forensics framework.

Cem Gurkok submitted new analysis capabilities to the 2013 and 2014 Volatility plugin contests [43]. These plugins focused on detection of kernel level rootkits that Volatility did not detect at the time. This included DTrace hooks, inline code hooks, and malicious TrustedBSD policy handlers. These research efforts have largely been incorporated into the stable Volatility code repository and releases.

In order to recovery Apple KeyChain encryption keys from memory, the original Volafox developer, Kyeong-Sik Lee, created a Volafox plugin that could extract potential KeyChain encryption keys from a memory sample. He also created a standalone tool, chainbreaker, that would ingest the potential set of keys, bruteforce each one, and display the contents of the keychain upon successful decryption [44]. In another effort to break OS X encryption, in early 2016 Thomas White created a Volatility plugin that can extract FileVault2 encryptions from memory captures [45].

During the course of my master's program, I, along with my advisor, Dr. Golden Richard, published two papers relating to OS X memory forensics. The first, which was published at DFRWS 2014 and won the best paper award, described how compressed, in-memory swap stores could be analyzed in order to recover a wealth of forensic data. Since this store is compressed in-memory, traditional methods of memory forensics, such as scanning and regular expression search, would miss any contained artifacts. The plugins described in this paper allowed for decompression of such pages, leading to full analysis [45]. The second paper discussed detection of kernel level rootkits through multiple methods in which existing memory forensics tools would miss [46]. The rootkits discussed in this paper could steal a variety of data related to a system's users as well as hide malicious activity from live system inspection. All of the techniques disclosed in this paper were researched and developed as Volatility plugins.

2.3. Objective-C Security Analysis

In 2015, a researcher with the handle "nemo" published a paper, "Modern Objective-C Exploitation Techniques" in the Phrack journal [10]. In this paper, a view of Objective-C classes and runtime data structures as they are stored in memory is presented. Although nemo's analysis was not conducted for the same reasons as ours, many of the data structures discussed in the Phrack article are the same as those needed for the research presented in this paper.

2.4. Userland Runtime Analysis

Much like Objective-C for OS X and iOS, Google provides a dedicated runtime for applications on its Android platform. Known as Dalvik, this runtime provides a rich set of consistent APIs for accessing the hardware and software components of Android devices. Also, like Objective-C, a wide range of malware samples has abused Dalvik and its features.

To allow malware analysts to deeply explore Dalvik and its runtime state, a number of techniques have been developed. The first was presented at Source Seattle 2011 [33]. In that work, an algorithm for locating all of Dalvik's classes in memory along with their associated methods and instance variables was presented. This included the ability to present the human-readable form of variables, such as the readable characters for string types and the numerical values for integer types. No source code was ever released, however.

In 2013, Holger Macht published his Master's thesis titled "Live Memory Forensics on Android with Volatility". His thesis provides precise details of Dalvik's data structures in memory as well as a number of Volatility plugins to find and analyze all of the loaded classes [34]. This level of detail allows investigators to immediately find all data structures related to a malware sample as well as locate its code in memory.

These previous efforts for Dalvik closely mirror the goals of the research for the Objective-C runtime.

2.5. Userland Malware Detection

A Volatility developer, Michael Ligh, released a set of plugins to analyze a number of Microsoft Windows userland APIs that provide functionality for DLL injection, keystroke logging, function hooking, and more. These were documented on the Volatility Labs blog [11, 12, 13, 14] as well as reproduced in greater detail in the book The Art of Memory Forensics [15].

Although the data structures and algorithms discussed in this thesis are completely different from the ones discussed in Ligh's work, the work of this thesis was influenced by his, as many of the same abuses can also be performed against OS X systems.

3. Objective-C

3.1. Background

Objective-C is an open source [21] language and associated runtime maintained by Apple for developers on the OS X and iOS platforms. Objective-C abstracts away many of the difficult aspects of programming systems software in C and C++ while still retaining many of the familiar semantics. The runtime provides very flexible runtime support for function calls, class instantiation, and use of variables and class members. For instance, all class and class member accesses can be performed based on a string name at runtime. Similarly, any class can locate other classes and instances at runtime based on string descriptions. As described in Chapter 4, this dynamic runtime environment provides a wide range of features that can be abused by malware.

Of particular relevance for memory analysis, Objective-C on Mac OS X also provides a rich API to access user and system activity, hardware peripherals (web cameras, microphones, keyboard, mouse, etc.), and integrity monitoring facilities. Due to the ease in which malware developers can leverage Objective-C to implement a wide range of malicious activity portably across Mac OS X versions, a number of high profile malware samples have been discovered that abuse the Objective-C runtime. Chapter 4 discusses how a number of these features are implemented by the runtime, how malware abuses them, and how they can be detected through memory forensics.

3.2. Runtime Operations and Data Structures

In order to analyze the state of the Objective-C runtime inside of a particular process, the developed plugins must be able to enumerate all loaded classes as well as their state. This analysis begins by locating the *realized_class_hash* global variable of the Objective-C library (*libobjc*). Plugins currently locate this global variable by one of two methods. The simplest, for the instances in which plugins can enumerate symbols of the library, is to find it by directly processing the library's symbol table. This can either be done with the library file from disk or using Volatiltiy's Mach-o APIs to enumerate symbols from process memory or the in-memory file cache. If the address is gathered from a file on disk then the address must be passed to each Volatility plugin. If the address cannot be discovered by these means, e.g., when an investigator is only supplied a memory sample and the symbol table is not memory-resident, then the Volatility plugins will scan through process memory and automatically locate the table.

The realized classes hash table holds a reference to every Objective-C class (type *objc_class*) loaded within a particular process. Of interest to us is that each class holds a reference to its members, including their name, type, and implementation pointer, its super classes, and its instance variables' definitions.

3.3. Incorporating into Volatility's Type System

In order for the Volatility plugins developed during this thesis to flexible and portable across versions, a representation of the relevant Objective-C data structures needed to be created in the Volatility types (*vtypes*) format. The vtypes format represents all possible C data structures and types as a Python hash table consumable by Volatility's core components. vtype specifications can be created manually during the course of reverse engineering, or, for target subsystems that are open source or for which debug symbols are available, automatically generated. Once a vtype specification is created for a target operating system or application version, then Volatility plugins can generically reference structure members and types, and Volatility's object system will transparently map the member to the type information for the correct target version.

Creating *vtypes* for Objective-C requires a hybrid approach, as although parts of Objective-C are open source, many of the components it links to are not. This prevents simply compiling a debug version of the Objective-C library and then automatically extracting the type information. Instead, a dummy application is used that references all the Objective-C data structures that Volatility relies on. This stripped down application can then be compiled with debugging information enabled and the types extracted using the built-in dwarfdump command. DWARF is the debug information format for ELF files (Linux) as well as for Mach-o files, which is the default OS X executable format. The output of dwarfdump can then be converted by existing leveraging existing Volatility helper code that puts the converted dwarfdump output into the vtype format.

4. Objective-C Malware

This chapter discusses three of the most popular methods by which Objective-C's runtime is abused by malware on Mac OS X.

4.1. Keystroke Logging

4.1.1. Background

Objective-C on Mac OS X provides two library functions for monitoring a system's keyboard [22]. The first, *addGlobalMonitorForEventsMatchingMask*, allows registration of a callback that will be executed each time a keystroke is pressed in any process other than the calling process. The second, *addLocalMonitorForEventsMatchingMask*, registers a callback for keystrokes pressed in the calling process. These can be used in combination when malware injects itself into a foreign, long-lived process that it wishes to monitor, along with all the other processes that are running.

4.1.2. Runtime Implementation

Both of the functions discussed above for registering a keyboard callback are implemented in the closed source AppKit framework. AppKit in turns relies on the HIToolbox sub-framework of the closed source Carbon framework in order to register the events with the global system monitor. When using these APIs, the caller must specify a handler, which will be called upon each key press, as well as an event mask, which specifies which events the user is interested in. The code in Figure 1 illustrates a simple keylogger using the global monitoring API to watch for keyboard down events, logging each keystroke to the system log.

```
-(void)applicationDidFinishLaunching:
 (NSNotification *)aNotification {
  [NSEvent
    addGlobalMonitorForEventsMatchingMask:
       NSKeyDownMask
         handler:^(NSEvent *event){
           NSLog(@"User pressed: %@",
             event.characters);
         }
   ];
}
```

Figure 1. Registering a global keylogger using Objective-C.

Through a reverse engineering effort, it was determined that to start the global registration process, *addGlobalMonitorForEventsMatchingMask* creates an instance of *NSGlobalEventObserver*. Both *NSGlobalEventObserver*, which is used for global monitoring, as well as

NSLocalEventObserver, which is used for same-process monitoring, inherit from *NSEventObserver*. This parent class has members *block* and *mask*, which are initialized using the function's parameters. *addGlobalMonitorForEventsMatchingMask* then calls *InstallEventHandler* [37] with a target parameter of *GetEventMonitorTarget()* and a handler_*GlobalObserverHandler*. It also sets the *userData* parameter to the *NSEventObserver* class that was previously created. *GetEventMonitorTarget* is a privileged, global event target that provides access to keyboard events. In Objective C, event targets are registered to receive events from the low-level hardware subsystems and are registered and handled by the runtime upon initialization. The *userData* parameter specifies a pointer to a function that will be sent to the initial handler of events, which in this case is _*GlobalObserverHandler*. Every time a key is pressed, *GlobalObserverHandler* then extracts the pointer to each user-defined callback and calls it with the key pressed.

4.1.3. Volatility Analysis Plugin

The *mac_observers* plugin was created to detect applications and libraries that have registered Objective-C callbacks using the two previously described APIs. It accomplishes this by finding every instance of *NSEventObserver*, and then reporting its handler address and event mask. The logic for this plugin is as follows:

Enumerate every process that maps the Objective-C library.
Locate the *objc_class* structure for *NSEventObserver* by enumerating *realized_class_hash.*
Scan the data (read/write) memory regions of the process looking for the address of the class. This uses the fact that each instance of a class is represented by an *Object* structure whose first member, *isa*, points to its defining class. This successfully locates all instances of a given class.
For each instance found, its *handler* member is mapped to its backing file, if any, and the *mask* member bitmask is decoded into its human-readable event types.

Figure 2 shows the output of this plugin running against a sample keylogger application (kl) that implements the code shown in Figure 1.

```
$ python vol.py -f kl.raw mac_observers
Volatility Foundation Volatility Framework 2.5
Name Pid  Class                Mask      Method Address    Library
---- ---  -------------------------- ---------  ------------------ -----------
kl  943  _NSGlobalEventObserver   NSKeyDown  0x0000000100001390 /Users/b/kl
```

Figure 2. Output of the new Volatility mac_observers plugin, which detects keystroke loggers

As Figure 2 illustrates, the handler application (kl) is correctly discovered, as is the fact that kl has registered interest in key down events. These events fire immediately after a key is pressed. Please note that the *mask* parameter for the Objective-C APIs described allows for not only monitoring the keyboard, but also mouse clicks and presses of a touch-screen device. The plugin properly decodes the mask to uncover all of these event types.

4.2. Method Swizzling

4.2.1. Background

Objective-C provides the ability for user-defined classes to "swizzle" methods of other classes loaded within the runtime. Swizzling a method involves swapping the method's implementation dynamically at runtime with that of another implementation. Future calls to a swizzled method use the new implementation instead of the original. Swizzling essentially allows dynamic updates to method implementations, including those that might otherwise be very difficult to modify, e.g., methods for which no source code is available.

From a malware analysis perspective, this is very similar to API hooking, which has been implemented in numerous malware samples across all modern operating systems. Traditionally, API hooks are detected by looking for functions whose first several bytes have been overwritten (i.e., evidence *inline hooks*), as well as examining runtime tables used to map function names to their runtime addresses for anomalies. These traditional hooks are already detected on Windows through Volatility's *apihooks* plugin [24] and on Mac through the *mac_apihooks* plugin [25].
Unfortunately, all existing methods for detecting API hooks will completely miss method swizzling in Objective C applications, since the call redirection is implemented inside the language runtime, and not through manipulation of the dynamic loader. This means that currently memory analysis cannot be used to detect malware that is deploying swizzling, nor will any information be provided about which hooks have been installed.

The most infamous malware to use method swizzling was Crisis [23]. Although this rootkit was recently shown to be detectable by memory analysis techniques [35], only the kernel components of the malware were detected. To the best of my knowledge, no publicly available memory analysis research has been presented that proposes techniques for detecting the Objective-C components of Crisis (or of any other Objective-C based malware). As discussed in [29], Crisis leverages method swizzling for a number of purposes including hiding processes from Apple's Activity Monitor, taking screenshots of infected systems, activating and recording web cameras and microphones, and hooking a wide variety of browser activity. It also employs methods for evading antivirus protection. This is particularly concerning as OS X is used almost exclusively on end-user systems, and tools like Crisis are used to target individuals of interest to both government and criminal organizations.

4.2.2. Runtime Implementation

Method swizzling is accomplished at runtime by calling the *method_exchangeImplementations* function [26]. This function takes two parameters, the first being a reference to the original method to be swizzled and the second a reference to the replacement method. Each method is specified by its string-based name. In order to get a reference to a particular method of a particular class, the *class_getInstanceMethod* function can be used. This function takes a reference to a class and the string name of a method and

returns its reference. To get a reference to a particular class, the *objc_getClass* function can be called with the first parameter set to the string name of the class. The code snip in Figure 3 illustrates how Crisis performs these operations to hook the Safari web browser.

From code injected into the Safari process, Crisis locates the *BrowserWindowController* class through *objc_getClass*. It then calls its own *swizzleMethod* function, passing the class, the Safari *webFrameLoadCommitted,* method and the *webFrameLoadCommittedHook* method, defined by Crisis. This allows Crisis to intercept every call to the method *webFrameLoadCommitted.*

```
className = objc_getClass("BrowserWindowController");

swizzleMethod(className,
 @selector(webFrameLoadCommitted:),
 className,
 @selector(webFrameLoadCommittedHook:));

function swizzleMethod(c1, m1, c2, m2) {
        method_exchangeImplementations(
         class_getInstanceMethod(c1, m1),
         class_getInstanceMethod(c2, m2));
}
```

Figure 3. Excerpt of Crisis' Hooking Code

Runtime-supported swizzling makes method replacement at runtime trivial, as Objective-C can locate the original class in memory and then provide functionality to exchange the method's implementation in a safe and consistent manner. This is much simpler than traditional API hooks that require malware to overwrite potentially running code or to manually tamper with the dynamic loader's runtime data structures.

Internally, to install the new implementation method in a swizzling operation, the Objective-C runtime locates the *method_t* structure corresponding to the method in the given class. Each class's members are stored in a list pointed to by the *bits* member of the class. Once the method structure is located, the runtime then sets the *imp* method of the corresponding *method_t* structure to the new implementation. The *imp* member is simply a pointer to the beginning of the code (instructions) for the method.

4.2.3. Volatility Analysis Plugin

The *mac_swizzled* plugin was created to detect swizzled Objective-C methods. By default, the plugin will:

Enumerate every process that maps the Objective-C library.
Locate all classes using either the given *realized_class_hash* address or by scanning.
For each class found, enumerate every method.
Print the method along with its address in memory and backing library, if any.

Figure 4 illustrates the output of the *mac_swizzled* plugin, using the default output (the pathnames have been trimmed in the figure to make them fit). As the figure shows, the plugin is able to successfully locate and print information about all loaded methods. This can be very useful when an analyst wants to fully understand what is occurring on a system and all the components loaded into a particular process. A downside of this approach, however, is that it produces hundreds of lines of output per process. This prevents effective use of the plugin in a triage effort by an analyst working a real incident. To help in such situations, the plugin also provides a *--triage* option that only outputs methods that meet one or more criteria. This is similar to the *alertMsg* function of RegRipper as implemented by Harlan Carvey [27].

The first alert type is generated if a method is implemented in a different library than the majority of the other methods of the class. This is accomplished by keeping a hash table of each class and the libraries its methods use. Once enumeration is completed, the libraries used by each class are compared to ensure that all methods of each class are implemented in the same source. From my study of real-world and proof-of-concept malware, one method being swizzled is enough to accomplish specific malicious tasks. This makes the alert very effective against real-world samples.

The second alert triggers if swizzled methods point to anonymous (non-file backed) regions. Using the default runtime API, all class method implementations should be in a process region backed by the implementing library. In the case of shellcode or reflective library injection [28] though, the method implementation will reside within an anonymous memory region. This again makes for simple alerting logic. The last alert type reports if a method is implemented in a library loaded from a suspicious directory, such as */tmp* or */private/var/tmp*.

```
$ python vol.py -f memdmp.raw mac_swizzled -p 1497

Name Pid  Class       Method      Method Address    Library
---- ---- ------------- ------------ ------------------ -------
kl  1497 NSInputManager dealloc      0x00007fff95ba9d7f /System/Library/.../Versions/C/AppKit
kl  1497 NSInputManager finalize     0x00007fff95ba9ead /System/Library/.../Versions/C/AppKit
...
kl  1497 NSInputManager description  0x00007fff95ba9f5c /System/Library/.../Versions/C/AppKit
kl  1497 NSInputManager image        0x00007fff95ba9d6e /System/Library/.../Versions/C/AppKit
kl  1497 NSInputManager isEnabled    0x00007fff95ba9a62 /System/Library/.../Versions/C/AppKit
...
kl  1497 NSInputManager hasMarkedText 0x00007fff95baa235 /System/Library/...//Versions/C/AppKit
kl  1497 NSInputManager selectedRange 0x00007fff95baa2e5 /System/Library/...//Versions/C/AppKit
kl  1497 NSInputManager insertText:  0x00007fff95ba9fe0 /System/Library/...//Versions/C/AppKit
```

Figure 4. Output of the new Volatility *mac_swizzled* plugin, which detects Objective-C pointer swizzling.

Combined, these filtered alerts provide investigators with immediately actionable indicators as opposed to hundreds of data points that must be manually filtered.

4.3. Named Ports

4.3.1. Background

Objective-C provides the ability for applications to register ports that are then accessible to all other Objective-C applications, to provide inter-process communication. This is handled by the *NSPortNameServer* class [30], which interacts with the Distributed Object subsystem [31]. Crisis leverages this functionality in order to mark a system as infected. Since

Crisis injects itself into many processes, it needs a method to ensure that different processes do not all attempt to infect the system and leave it in an inconsistent state. Figure 5 illustrates the named port check in Crisis.

```
if (![[NSPortNameServer
   systemDefaultPortNameServer]
   registerPort: port
        name: @"com.apple.mdworker.executed"]) {
   errorLog(@"NSPort check error! Backdoor
         is already running");
  exit(-1);
}
```

Figure 5. Crisis' registration named port check.

In this code, Crisis attempts to register the "com.apple.mdworker.executed" named port. The function will fail if the port is already registered, which allows Crisis to detect the previous installation of the backdoor.

This use of a global system infection marker is analogous to the well-documented behavior of Windows malware samples that leverage mutexes or atoms to mark a system as infected. In facts, building a dictionary of known-bad mutexes and atom strings to immediately identify malware is a technique used by many forensics analysts. Similarly, experienced security teams will build whitelist of mutexes from a known-good copy of a system so that they can then later be used to immediately spot anomalies in future investigations. Similar approaches can be ported to OS X systems to spot both known and unknown malware samples.

4.3.2. Runtime Implementation

On OS X versions 10.6 through 10.9, registered ports are stored in a hash table of the calling process' associated *launchd* process. Depending on the OS version and system runtime state, there may only be one *launchd* process, run as *root* (UID 0), or there may be several *launchd* processes. In the case of multiple *launchd* instances, there is generally one per user login as well as for specific services, such as the file system indexer, *Spotlight*.

```
$ python vol.py  -f infected-with-crisis.raw  mac_launchd_ports
Volatility Foundation Volatility Framework 2.5
Pid      Address            Name
-------- ------------------ ----
    1    0x0000000010443ab60  com.apple.security.pboxd
    1    0x0000000104428a80  com.apple.SystemConfiguration.PPPController
    1    0x0000000010441cb90  com.apple.sandboxd
...
  119    0x0000000104e15820  com.apple.pictd
  119    0x0000000104e0db80  com.apple.dock.appstore
  119    0x0000000104e07800  com.apple.mdworker.prescan.0
  119    0x0000000104e25980  com.apple.mdworker.executed
  119    0x0000000104e08330  com.apple.axserver
  119    0x0000000104e1d630  com.apple.syncdefaultsd.push
  119    0x0000000104e162a0  com.apple.printtool.agent
...
```

Figure 6. Output of the new Volatility mac_launchd_ports plugin, which analyzes the use of named ports on Mac OS X.

This hash table is a global variable named *port_hash*. Each key of the hash table is a structure of type *machservice, which has* two members of interest. The first, *port_hash_sle,* is the structure's linkage into the per-hash bucket list of services. The second member of interest is *name*, which contains the ASCII name of the service. In the case of the port registered by Crisis, the *name* member is the NULL-terminated string "com.apple.mdworker.executed". This hash table is populated through a client process, such as Crisis, by calling the *register-Port* API. Internally, the port is represented by a *NSMachBootstrapServer* instance. This class is implemented in the proprietary OS X *Foundation* framework. Binary analysis of this class' implementation reveals that it communicates with the associated remote *launchd* process through a call to *bootstrap_look_up2.* This function is implemented inside of the open source *liblaunchd*, which clients link with in order to use *launchd*'s client API. Through OS X's IPC API, *liblaunchd* calls its server component (*job_mig_look_up2*) inside the remote *launchd* process. This remote function then checks if the port is already registered, and if not, it adds it to *port_hash,* among other initialization tasks.

Beginning with OS X 10.10 (Yosemite), Apple closed source *launchd* and moved it to the proprietary *libxpc* library. Currently, I have not performed analysis on the newer implementation, since Jonathan Levin, a well-known OS X and iOS researcher, has claimed that he has reverse-engineered the entire *libxpc*, and will be releasing a complete, open source clone with his book in early 2016 [36]. When his open source implementation is released, I will then add support for the newer OS X versions to the new plugin, which is described next.

4.3.3. Volatility Analysis Plugin

In order to analyze registered ports for *launchd* instances, the *mac_launchd_ports* Volatility plugin was developed. The plugin works as follows:

Enumerate all processes and filter to *launchd* instances.
Find where *launchd* is mapped into process memory by walking the process memory mappings.
Locate *port_hash* through a given command line option or by scanning. Similar to finding *realized_class_hash*, the offset of this symbol can be found manually from the file on disk or through inspection of */sbin/launchd,* extracted from the in-memory file cache. Volatility also supports dynamically finding it if the symbol table is memory resident.
Walk each index of *port_hash* (maximum of 32), and each linked list stored at each index.
Print the process ID, address, and name of each registered Mach service.

The plugin also makes a best effort to filter out corrupt data, which is often encountered during memory forensics of real systems. It does this by ensuring that pointers point to valid addresses (i.e., are present in memory) as well as validates that the *name* member contains a valid ASCII string.

Figure 6 illustrates the output of the *mac_launchd_ports* plugin executed against a memory sample infected with Crisis. In the output, along with benign ports, the one that Crisis registers is also evident.

Using a known-bad set of named ports would allow immediate identification of malware like Crisis. An investigator could also build a whitelist of named ports from a known-good system and then use it to quickly find named ports of forensic interest.

5. Conclusions and Future Work

In this thesis, new techniques for detecting userland malware written in Objective-C for Mac OS X were presented. This research effort involved a deep analysis of the Objective-C runtime and APIs, to identify interesting process state that is potentially indicative of malicious behavior, such registration of keystroke event monitors, the use of named ports, and pointer swizzling. The plugins created for the Volatility framework automatically analyze important artifacts in the Objective-C runtime and produce output that can easily be used by analysts to isolate and more deeply investigate these behaviors. Existing approaches for malware detection on Mac OS X do not detect the targeted behaviors, since the Objective-C runtime maintains state outside of the dynamic loader and the code section of executables.

With the rapid adoption of OS X systems in corporate and government networks, along with the increasing number of advanced OS X malware samples already found in the wild, the need for robust detection of OS X specific rootkits will continue to grow. By incorporating Objective-C inspection techniques into their investigative workflows, forensic analysts will be far better prepared to detect and analyze advanced threats.

In order to stay ahead of possible malware infection vectors, I plan to further explore the Objective-C runtime to find additional features and APIs that can be abused by malware. Because of the robust nature of the Objective-C runtime, I strongly suspect that additional work is needed to identify features that malware may leverage to operate undetected.

6. References

[1] “Kernel-Mode Code Signing Requirements” https://msdn.microsoft.com/en-s/library/windows/hardware/ff548239(v=vs.85).aspx. (Accessed January 25, 2016).
[2] Pot, J., “What Mac Users Need To Know About El Capitan Security” http://www.makeuseof.com/tag/mac-security-el-captan-rootless/. (Accessed January 25, 2016).
[3] “Kernel Patch Protection” https://en.wikipedia.org/wiki/Kernel_Patch_Protection. (Accessed January 25, 2016).
[4] "Defeating Windows Driver Signature Enforcement #1: Default Drivers," http://j00ru.vexillium.org/?p=1169 (Accessed January 25, 2016).
[5] Skape and Skywing, "Bypassing PatchGuard on Windows x64," http://www.uninformed.org/?v=3&a=3. (Accessed January 25, 2016).
[6] "Analyzing the Uroburos PatchGuard Bypass," https://blogs.mcafee.com/mcafee-labs/analyzing-uroburos-patchguard-bypass/. (Accessed January 25, 2016).
[7] "Breaking OS X Signed Kernel Extensions with the NOP," https://reverse.put.as/2013/11/23/breaking-os-x-signed-kernel-extensions-with-a-nop/. (Accessed January 25, 2016).
[8] Erwin, D., "Ventir Trojan Intercepts Keystrokes from Mac OS X Computers," https://www.intego.com/mac-security-blog/ventir-trojan-intercepts-keystrokes-from-mac-os-x-computers/. (Accessed January 25, 2016).
[9] Katsuki, T. "Crisis: The Advanced Malware," http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/crisis_the_advanced_malware.pdf (Accessed January 25, 2016).
[10] nemo, "Modern Objective-C Exploitation Techniques," http://www.phrack.org/papers/modern_objc_exploitation.html. (Accessed January 25, 2016).
[11] Ligh, M. H., "MoVP 1.1 Logon Sessions, Processes, and Images," http://volatility-labs.blogspot.com/2012/09/movp-11-logon-sessions-processes-and.html. (Accessed January 25, 2016).
[12] Ligh, M. H., "MoVP 1.2 Window Stations and Clipboard Malware," http://volatility-labs.blogspot.com/2012/09/movp-12-window-stations-and-clipboard.html. (Accessed January 25, 2016).
[13] Ligh, M. H., "MoVP 2.2 Malware In Your Windows," http://volatility-labs.blogspot.com/2012/09/movp-22-malware-in-your-windows.html. (Accessed January 25, 2016).
[14] Ligh, M. H., "OMFW 2012: Malware In the Windows GUI Subsystem," http://volatility-labs.blogspot.com/2012/10/omfw-2012-malware-in-windows-gui.html. (Accessed January 25, 2016).
[15] Ligh, M.H., Case, A., Levy, J. and Walters, A., 2014. The Art of Memory Forensics. Indianapolis, ID: Wiley.
[16] Petroni, N.L., Walters, A., Fraser, T. and Arbaugh, W.A., "FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory," *Digital Investigation* 3(4), 2006, pp.197-210.
[17] Kornblum, J., "Using Every Part of the Buffalo in Windows Memory Analysis." *Digital Investigation* (4)1, 2007, pp. 24-29.

[18] Cohen, M., "Forensic Analysis of Windows User space Applications through Heap allocations," *Proceedings of the 3rd IEEE International Workshop on Security and Forensics in Communication Systems*, 2015.
[19] Richard, G. G., and Case, A. "In Lieu of Swap: Analyzing Compressed RAM in Mac OS X and Linux." *Digital Investigation* 11 (2014): S3-S12.
[20] https://channel9.msdn.com/Blogs/Seth-Juarez/Memory-Compression-in-Windows-10-RTM/. (Accessed January 25, 2016).
[21] https://opensource.apple.com/. (Accessed January 25, 2016).
[22] "NSEvent Class Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ApplicationKit/Classes/NSEvent_Class/. (Accessed January 25, 2016).
[23] Vilaca, P. "Tales from Crisis, Chapter 3: the Italian Rootkit Job", http://reverse.put.as/2012/08/21/ tales-from-crisis-chapter-3-the-italian-rootkitjob/. (Accessed January 25, 2016).
[24] "Volatility *apihooks* Plugin," https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/apihooks.py. (Accessed January 25, 2016).
[25] "Volatility *mac_apihooks* Plugin," https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/mac/apihooks.py. (Accessed January 25, 2016).
[26] "Mac OS X Objective-C Runtime Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/#//apple_ref/c/func/method_exchangeImplementations. (Accessed January 25, 2016).
[27] "regripper tool", http://windowsir.blogspot.com/2013/04/regripper-updates.html. (Accessed January 25, 2016).
[28] skape and Turkulainen, J., "Remote Library Injection," http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf. (Accessed January 25, 2016).
[29] Nayyar, H. "An Opportunity in Crisis," https://www.sans.org/reading-room/whitepapers/threats/opportunity-crisis-34600. (Accessed January 25, 2016).
[30] "Foundation Framework Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSPortNameServer_Class/#//apple_ref/occ/instm/NSPortNameServer/. (Accessed January 25, 2016).
[31] "Introduction to Distributed Objects," https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/DistrObjects/DistrObjects.html#//apple_ref/doc/uid/10000102i. (Accessed January 26, 2016).
[32] "Volatility Memory Analysis Framework," https://github.com/volatilityfoundation/volatility. (Accessed January 26, 2016).
[33] Case, A. "Memory Analysis of the Dalvik (Android) Virtual Machine," http://www.slideshare.net/AndrewDFIR/android-memoryanalysis (Accessed January 26, 2016).
[34] Macht, H. "Live Memory Forensics on Android with Volatility," https://www1.informatik.uni-erlangen.de/filepool/publications/Live_Memory_Forensics_on_Android_with_Volatility.pdf, M.S. Thesis, Department of Computer Science, Friedrich-Alexander University Erlangen-Nuremberg (Accessed January 26, 2016).

[35] Case, A., and Richard, G. G., "Advancing Mac OS X rootkit detection." *Digital Investigation* 14 (2015): S25-S33.
[36] J. Levin, http://newosxbook.com/articles/jlaunchctl.html (Accessed February 4, 2016).
[37] "Carbon Event Manager Programming Guide," https://developer.apple.com/legacy/library/documentation/Carbon/Conceptual/Carbon_Event_Manager/CarbonEvents.pdf (Accessed February 4, 2016).
[38] Suiche M. Advanced Mac OS X physical memory analysis. In: Blackhat DC security conference; 2010.
[39] Volafox memory analysis framework. 2015. https://code.google. com/p/volafox/.
[40] Case A. Mac memory analysis with Volatility. In: 2012 SANS DFIR Summit; 2012.
[41] Case A. Hunting mac malware with memory forensics. In: 2014 RSA USA conference; 2014.
[42] Hay A. Forensic memory analysis for Apple OS X (Master's thesis). Air Force University; 2011.
[43] Gurkok C. What's in your silicon?. 2015. siliconblade.blogspot.com/. Silicon Blade Blog
[44] "chainbreaker," https://github.com/n0fate/chainbreaker (Accessed March 18th, 2016).
[45] "EXTRACTING FILEVAULT 2 KEYS WITH VOLATILITY," https://tribalchicken.com.au/security/extracting-filevault-2-keys-with-volatility/ (Accessed March 18th, 2016).
[46] Richard GG, Case A. In lieu of swap: analyzing compressed RAM in Mac OS X and Linux. Digit Investigation 2014;11:S3e12.
[47] Case A, Richard GG. Advancing Mac OS X rootkit detection. In the proceedings of the 2015 Digital Forensics Research Workshop (DFRWS).

VITA

The author was born in Metairie, LA. He obtained his Bachelor's degree in Computer Science from the University of New Orleans in 2009. In 2013, he resumed classes to complete his Master's Degree. He is the Director of Research at Volexity, and a board member of The Volatility Foundation.