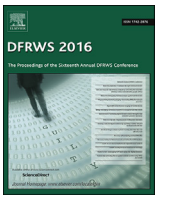
Digital Investigation 18 (2016) S3–S10

Contents lists available at ScienceDirect

Digital Investigation

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



DFRWS USA 2016 — Proceedings of the 16th Annual USA Digital Forensics Research Conference

Detecting objective-C malware through memory forensics

Andrew Case [a], Golden G. Richard III [b, *]

[a] Volexity, United States
[b] Dept. of Computer Science, University of New Orleans, LA 70148, United States

Keywords:
Memory forensics
Malware
Mac OS X
Objective-C

A B S T R A C T

Major advances in memory forensics in the past decade now allow investigators to efficiently detect and analyze many types of sophisticated kernel-level malware. With operating systems vendors now routinely enforcing driver signing and integrating strategies for protecting kernel data, such as Patch Guard, userland attacks are becoming more attractive to malware authors, as evidenced in the notorious Crisis malware. We therefore turn our attention to improving memory forensics techniques for analysis of malware in userland. In this paper, we focus on new methods for detecting userland malware written in Objective-C on Mac OS X. As the paper illustrates, Objective-C provides a rich set of APIs that malware can use to manipulate and steal application data and to perform other malicious activities. Our novel memory forensics techniques deeply examine the state of the Objective-C runtime inside of targeted processes, identifying a number of suspicious activities, from keystroke logging to pointer swizzling. We then examine our techniques against memory samples infected with malware found in targeted OS X attacks.

Introduction

Memory forensics has quickly become one of the primary methods for digital forensic investigators to detect and analyze sophisticated malware and rootkits. Since memory forensics tools are minimally reliant on the running operating system for gathering volatile data, they can often locate and analyze forensics artifacts that live analysis tools would miss. 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 (Kernel-Mode Code) and Mac OS X (Pot) as well as Microsoft Patch Guard (Kernel Patch). 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 (Defeating Windows Driver; Skape and Skywing; Uroburos PatchGuard; Breaking OS).

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

* Corresponding author. Dept. of Computer Science, Univ. of New Orleans, New Orleans, LA 70148, United States.
E-mail addresses: andrew@dfir.org (A. Case), golden@cs.uno.edu (G.G. Richard).

http://dx.doi.org/10.1016/j.diin.2016.04.017

data, and maintaining long-term persistence, without having to enter kernel mode. On Windows, this has led to malware with a single executable 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 (Erwin) and Crisis (Katsuki), 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.

In this paper, we explore 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 paper, 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, we researched and developed novel 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 (Volatility Memory). 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 Section 5 of this paper, each of our developed techniques is presented along with a newly created Volatility plugin that implements the described analysis. Upon publication of the paper, the plugins will be contributed to the open source Volatility project for use by the forensics community.

Related work

Wardle (2014) provides a look at several examples of Mac OS X malware and surveys persistence mechanisms. Although no previous memory forensic analysis efforts exist for deep analysis of Objective-C applications on Mac OS X, there has been substantial work in a number of related areas. These efforts are discussed below.

Objective-C security analysis

In 2015, a researcher with the handle “nemo” published a paper, “Modern Objective-C Exploitation Techniques” in the Phrack journal (nemo). 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.

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 by Andrew Case and presented at Source Seattle 2011 (Case). 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 (Macht). 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 our research for the Objective-C runtime.

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 (Ligh and MoVP 1.1; Ligh and MoVP 1.2; Ligh and MoVP 2.2; Ligh) as well as reproduced in greater detail in the book The Art of Memory Forensics (Ligh et al., 2014).

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

Objective-C

Background

Objective-C is an open source (opensource.apple.com) 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 Section 5, 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. In Section 5 we discuss how a number of these features are implemented by the runtime, how malware abuses them, and how they can be detected through memory forensics.

Runtime operations and data structures

In order to analyze the state of the Objective-C runtime inside of a particular process, our techniques 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*). We currently locate this global variable by one of two methods. The simplest, for the instances in which we 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 our 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.

Objective-C malware

In this section we discuss three of the most popular methods by which Objective-C's runtime is abused by malware on Mac OS X.

Keystroke logging

Background

Objective-C on Mac OS X provides two library functions for monitoring a system's keyboard (NSEvent). 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.

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 Fig. 1 illustrates a simple keylogger using the global monitoring API to watch for keyboard down events, logging each keystroke to the system log.

Through a reverse engineering effort, we 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* (Carbon Event Manager) 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.

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:

1) Enumerate every process that maps the Objective-C library.
2) Locate the *objc_class* structure for *NSEventObserver* by enumerating *realized_class_hash*.
3) 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.
4) 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.

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

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

Fig. 2 shows the output of this plugin running against a sample keylogger application (kl) that implements the code shown in Fig. 1. As Fig. 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. We note that the *mask* parameter for the Objective-C APIs we have 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.

Method swizzling

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 (Volatility apihooks Plugin) and on Mac through the *mac_apihooks* plugin (Volatility mac_apihooks Plugin).

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.

The most infamous malware to use method swizzling was Crisis (Vilaca). Although this rootkit was recently shown to be detectable by memory analysis techniques (Case and Richard, 2015), only the kernel components of the malware were detected. To our best 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 (Nayyar) and confirmed through our own research, 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 malware like Crisis is used to target individuals of interest to both government and criminal organizations.

Runtime implementation

Method swizzling is accomplished at runtime by calling the *method_exchangeImplementations* function (Mac OS). 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 Fig. 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*.

Runtime-supported swizzling makes method replacement at runtime trivial, as Objective-C can locate the original class in memory and then provide functionality to

```
$ 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
```

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

```
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));
}
```

Fig. 3. Excerpt of Crisis' hooking code.

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.

Volatility analysis plugin

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

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

Fig. 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, our 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 (regripper tool).

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 our 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 (skape and Turkulainen) 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*.

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

Named ports

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 (Foundation Framework Reference), which interacts with the Distributed Object subsystem (Distributed Objects). 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. Fig. 5 illustrates the named port check in Crisis. 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 fact, 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.

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 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
```

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

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

Fig. 5. Crisis' named port registration check.

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 *registerPort* 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, we 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 in the new edition of his book (Levin). When his open source implementation is released, we will then add support for the newer OS X versions to our new plugin, which is described next.

Volatility analysis plugin

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

1) Enumerate all processes and filter to *launchd* instances.
2) Find where *launchd* is mapped into process memory by walking the process memory mappings.
3) 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.
4) Walk each index of *port_hash* (maximum of 32), and each linked list stored at each index.
5) 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 optionally validating that the *name* member contains a valid ASCII string.

Fig. 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.

Conclusions and future work

In this paper, we presented new techniques for detecting userland malware written in Objective-C for Mac OS X. Our work involved a deep analysis of the Objective-C

```
$ python vol.py  -f infected-with-crisis.raw  mac_launchd_ports
Volatility Foundation Volatility Framework 2.5
Pid        Address                  Name
--------   ---------------------    ----
       1      0x000000010443ab60   com.apple.security.pboxd
       1      0x0000000104428a80   com.apple.SystemConfiguration.PPPController
       1      0x000000010441cb90   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
…
```

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

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 we've 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 behaviors we have targeted, 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, our research team plans 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, we strongly suspect that additional work is needed to identify features that malware may leverage to operate undetected.

Finally, we are also investigating the Swift runtime, as this language is gaining momentum on both OS X and iOS.

References

"Analyzing the Uroburos PatchGuard Bypass," https://blogs.mcafee.com/mcafee-labs/analyzing-uroburos-patchguard-bypass/. [Accessed 25.01.16].

"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 25.01.16].

Case, A. "Memory Analysis of the Dalvik (Android) Virtual Machine," http://www.slideshare.net/AndrewDFIR/android-memoryanalysis [Accessed 26.0.1.16].

Case A, Richard GG. Advancing Mac OS X rootkit detection. Digit Investig 2015;14:S25–33.

"Carbon Event Manager Programming Guide," https://developer.apple.com/legacy/library/documentation/Carbon/Conceptual/Carbon_Event_Manager/CarbonEvents.pdf [Accessed 04.02.16].

"Defeating Windows Driver Signature Enforcement #1: Default Drivers" http://j00ru.vexillium.org/?p=1169. [Accessed 25.01.16].

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 25.01.16].

"Foundation Framework Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSPortNameServer_Class/#//apple_ref/occ/instm/NSPortNameServer/. [Accessed 25.01.16].

https://opensource.apple.com/. [Accessed 25.01.16].

"Introduction to Distributed Objects," https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/DistrObjects/DistrObjects.html#//apple_ref/doc/uid/10000102i. [Accessed 26.01.16).

Katsuki, T. "Crisis: The Advanced Malware," http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/crisis_the_advanced_malware.pdf [Accessed 25.01.16].

"Kernel Patch Protection" https://en.wikipedia.org/wiki/Kernel_Patch_Protection. [Accessed 25.01.16].

"Kernel-Mode Code Signing Requirements" https://msdn.microsoft.com/en-s/library/windows/hardware/ff548239(v=vs.85).aspx. [Accessed 25.01.16].

J. Levin, http://newosxbook.com/articles/jlaunchctl.html [Accessed 04.02.16].

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 25.01.16].

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 25.01.16].

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 25.01.16).

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 25.01.16).

Ligh MH, Case A, Levy J, Walters A. The art of memory forensics. Indianapolis, ID: Wiley; 2014.

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 26.01.16].

"Mac OS X Objective-C Runtime Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/#//apple_ref/c/func/method_exchangeImplementations. [Accessed 25.01.16].

Nayyar, H. "An Opportunity in Crisis," https://www.sans.org/reading-room/whitepapers/threats/opportunity-crisis-34600. [Accessed 25.01.16].

nemo, "Modern Objective-C Exploitation Techniques," http://www.phrack.org/papers/modern_objc_exploitation.html. [Accessed 25.01.16].

"NSEvent Class Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ApplicationKit/Classes/NSEvent_Class/. [Accessed 25.01.16].
Pot, J., "What Mac users need to know about EL Capitan security" http://www.makeuseof.com/tag/mac-security-el-captan-rootless/. [Accessed 25.01.16].
"regripper tool", http://windowsir.blogspot.com/2013/04/regripper-updates.html. [Accessed 25.01.16].
Skape and Skywing, "Bypassing PatchGuard on Windows x64," http://www.uninformed.org/?v=3&a=3. [Accessed 25.01.16].
skape and Turkulainen, J., "Remote Library Injection," http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf. [Accessed 25.01.16].
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 25.01.16].
"Volatility *apihooks* Plugin," https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/apihooks.py. [Accessed 25.01.16].
"Volatility *mac_apihooks* Plugin," https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/mac/apihooks.py. [Accessed 25.01.16].
"Volatility Memory Analysis Framework," https://github.com/volatilityfoundation/volatility. [Accessed 26.01.16].
Wardle P. Methods of malware persistence on Mac OS X. In: Proceedings of the virus bulletin conference; September, 2014.