9-1-2021

Modern macOS userland runtime analysis

Modhuparna Manna
LSU College of Engineering

Andrew Case
Volatility Foundation

Aisha Ali-Gombe
Towson University

Golden G. Richard
LSU College of Engineering

Recommended Citation

Manna, M., Case, A., Ali-Gombe, A., & Richard, G. (2021). Modern macOS userland runtime analysis. *Forensic Science International: Digital Investigation, 38* https://doi.org/10.1016/j.fsidi.2021.301221

Version of Record: https://www.sciencedirect.com/science/article/pii/S2666281721001293
Manuscript_194bf13deac90e3be5d92159465807bb

Modern macOS Userland Runtime Analysis

Modhuparna Manna[b,c], Andrew Case[a], Aisha Ali-Gombe[d], Golden G. Richard III[b,c]

[a] *Volatility Foundation*
[b] *Center for Computation and Technology, Louisiana State University*
[c] *School of Electrical Engineering & Computer Science, Louisiana State University*
[d] *Department of Computer and Information Science, Towson University*

Abstract

The continued rise of Apple's macOS in both the home and workplace has led to a significant rise in the capabilities of both malware and attacker toolkits that target the operating system and its users. Over the last several years there have been numerous documented instances of macOS users being targeted by governments, intelligence agencies, and criminal groups, and the end results of these attacks were the victims having highly sophisticated malware installed on their systems. Unfortunately, the rise of these threats has not been met with an equal research and development effort by the memory forensics community. This has led to a gap in automated analysis in memory forensic frameworks and has left inexperienced investigators with little chance of detecting the malware's presence. Even for experienced investigators, detection was still difficult in many circumstances and require significant manual investigation for a chance at success. This paper documents our research effort to close this gap through the development of novel memory forensic capabilities aimed at detecting advanced, real-world malware that targets macOS systems. This research was driven through analysis of numerous malware samples that were used as part of espionage and criminal attack campaigns, and the end result was three new Volatility plugins that automate the detection of such malware. By leveraging these plugins, investigators of all skill levels can detect macOS userland malware in an automated, scalable, and flexible manner.

Keywords: memory forensics, macOS forensics, memory analysis, malware, swift, objective c

1. Introduction

Over the last decade, memory forensics has emerged as a key component of modern computer defense. Memory forensics, which is the examination of volatile memory (RAM), provides analysts with a complete view of system state at the time of acquisition. By leveraging modern memory analysis frameworks, investigators can deeply explore the address space of the kernel and each process as well as historical data left behind in de-allocated pages. This level of insight allows much more thorough and fine-grained analysis compared to traditional analysis techniques, such as disk forensics and live system forensics, and it has led to memory analysis becoming a requirement of competent incident response workflows.

This rise of memory forensics has been driven by three main factors. The first is the sophistication of current malware, including the use of memory-only payloads and heavily obfuscated and encrypted code and data. The second factor is that the APIs provided by modern operating systems to control devices, such as web cameras and microphones, or to monitor system activity, such as keystrokes and user interaction, may be abused by malware to record microphone audio, access web camera streams, capture keystrokes, and covertly gain screenshots of sensitive data. Traditional analysis techniques are woefully inadequate against such malware and often force the analyst into manual malware analysis, which is both time and resource intensive. The final factor is the nearly ubiquitous use of encrypted file system data and network traffic by suspects of criminal investigations as well as malicious insiders. Combined, these factors necessitate memory forensics tools that can parse a wide range of subsystems in both kernel and userland memory to fully reconstruct system state.

While the field of memory analysis has seen significant research performed over the last decade, the bulk of this work has focused on the analysis of data structures in the kernel. This focus was driven by the need to detect kernel level malware as well as the fact that the data structures describing process data, networking artifacts, file system information, and many other common artifacts are stored within the kernel's address space. While the results of past research have led to significant kernel analysis capabilities, there is a severe lack of subsystem and runtime analysis capabilities in userland. Given the trend of all major operating system vendors towards strict-code signing requirements for 3rd-party kernel drivers, there have been significant developments in userland-only malware.

Email addresses: mmanna3@lsu.edu (Modhuparna Manna), andrew@dfir.org (Andrew Case), aaligombe@towson.edu (Aisha Ali-Gombe), golden@cct.lsu.edu (Golden G. Richard III)

Unfortunately, defensive research has not kept up with this trend, as existing memory analysis techniques generally only provide broad extraction of userland data, such as extracting an entire processes' address space or specific regions of a process. While these capabilities are certainly useful, they do not provide the structured analysis of activity that is needed to fully analyze running applications and code within process memory.

In this paper, we document our research effort to bridge this gap in a key userland subsystem - the Objective-C and Swift runtimes of Mac OS. These runtimes power a number of common Mac OS applications and are also the programming language of choice for a variety of malware samples and toolkits. Given the significant market share of Macs in both enterprise and home environments [1] as well as the targeting of Mac users by nation-state backed threat actors [2, 3, 4, 5, 6, 7], it is necessary for analysts to have full visibility of these runtimes. The sole research effort related to ours is largely outdated and only covered a few isolated portions of the runtime that were targeted by a specific malware sample. To advance the state of the art and to provide generic memory analysis capabilities against these runtimes, we examined a wide variety of runtime data structures and related APIs. We also deeply analyzed a number of OS X malware samples that abuse these runtimes. We then developed Volatility [8] plugins that automate the extraction of the structured runtime data. These plugins are configurable and support operations such as searching for known-bad classes and data, extracting all data matching a given pattern, and recognizing code patterns that are often abused for malicious purposes. The inclusion of these capabilities into memory forensic workflows will allow even novice investigators to automatically detect sophisticated real-world malware.

2. Related Work

2.1. Memory Analysis

The field of memory analysis has seen substantial research published over the last fifteen years. The 2005 DFRWS challenge [9] is widely considered to have been the catalyst for the initial research. This challenge required development of tools that could parse a Windows memory sample in a structured manner to recover key artifacts. Such tools did not exist at the time and led to several novel research efforts [10, 11]. In the following years, many new capabilities for Windows analysis were added, such as analysis of VAD trees [12], the registry in memory [13], and cached files [14]. Since then, there have been dozens of new capabilities added that are all now a part of standard Windows systems investigations. Structured analysis of Linux systems largely began with efforts to solve the 2008 DFRWS challenge DFRWS [15] and has seen considerable growth since then. Mainstream analysis of OS X samples began in 2013 [16] with the release of Volatility 2.3. Volatility [8] is the mostly wide used framework in the field and provides significant analysis capabilities for Windows, Linux, and macOS memory samples.

2.2. Userland Runtime Analysis

2.2.1. Android Runtime Analysis

There have been several efforts to analyze the data structures and effects of the Android application runtime. In [17], the Android runtime was analyzed to recover the classes loaded by an application as well as the metadata of those classes (instance variables, methods, etc.). In [18], the Android runtime was studied even further and algorithms were presented that allowed for location of class instances and decoding of variable data. In [19], a system was presented for recovery of heap metadata and objects, which led to recovery of a significant amount of application data.

2.2.2. Windows GUI Subsystem Analysis

The Volatility framework provides a suite of plugins to analyze the GUI subsystem of Windows systems such as wndscan, windows, wintree, messagehooks, etc. [20]. This allows detection of malware that leverages related APIs, such as for keystroke logging, monitoring of USB device insertion, and code injection. It also allows construction of user activity through mapping per-session data, reconstruction of screen views (screenshots), and the recovery of values from input forms and menus.

2.3. macOS Runtime Analysis

At DFRWS 2016, a paper was published that detailed Objective-C memory analysis [21]. The purpose of this effort was to develop Volatility plugins that would detect the Crisis malware [22] within memory samples. To accomplish this, select portions of the Objective-C runtime were analyzed to determine the address of class method handlers. This allowed detection of method swizzling as well as malware that registered keyloggers through the use of the *addGlobalMonitorForEventsMatchingMask* or *addLocalMonitorForEventsMatchingMask* functions.

While novel, this research is now largely outdated as it only supported analysis on macOS versions up to 10.9, which stopped being supported in 2016. Since then, six new versions of OS X have been released. Due to its age, the algorithm presented in this research for enumerating the loaded classes inside a process is no longer valid as well as the fact that it does not fully enumerate all methods associated with an object instance. Also, as mentioned in the future work portion of that research paper, there is no processing of Swift information, which leaves a large gap in analysis. Finally, the detection algorithms utilized by the plugins are very limited in their scope, and they leave much of the runtime information unexplored. For example, class methods are only examined on two criteria. The first is based on the method implementation's memory region, and it is examined for being mapped by a file in a suspicious directory or being in a non-file backed region.

The second criteria is if the region is hosting a handler for one of the previously mentioned event monitors. The use of these event monitors is not typical in current macOS malware and the referenced work is unable to detect the more popular and modern methods. This research also did not provide algorithms for decoding variable values, instead concentrating solely on enumerating names. This left behind strings and other data that may be useful when analyzing malware and investigating malicious insiders.

In this work, we contribute significant new capabilities for automating the examination of modern macOS userland runtimes. We begin by presenting not only enumeration of the variables of object instances, but also parsing of their values based on their types. This allows investigators to develop templates for repeatable analysis as well as automatically search for known indicators of malicious code. We then examine the classes loaded into a process in an attempt to find those often abused by malware. We end by showcasing our ability to deeply examine the in-memory code of class instances to discover the use of often-abused APIs. This allows detection of many keylogger types, of code that monitors the microphone and web camera stream, and much more. Combined, these capabilities provide generic detection of malicious macOS applications, and do not require malware sample-specific knowledge to detect suspicious behaviour.

3. Objective-C Memory Analysis

Objective-C is the primary userland runtime provided by Apple for development on its platforms [23]. It is features a C like syntax, but with features that allow for quicker and simpler development. Part of this feature set is a wide range of APIs that allow for direct interaction with numerous subsystems, including those that manage hardware devices and interface with users and user-driven activity. Given its prevalence on macOS systems along with its history of abuse by malware, being able to deeply analyze the runtime is necessary for complete investigations.

Providing full analysis of this runtime required several artifact recovery capabilities to be developed, including enumeration of the following:

- All loaded classes within a process
- The location of instances of each class
- The instance variables of each class
- The type-specific value of each instance's variables
- The methods of each class

Through recovery of this substantial dataset on a per-process basis, we were able to develop a variety of plugins that allow for full structured analysis of Objective-C. The remainder of this section describes how these goals were accomplished.

3.1. Enumerating Loaded Classes

Prior to macOS version 10.10, each loaded class was tracked by the *realized_class_hash* global variable. Starting with macOS 10.10 and continuing through the latest version, which is 10.15.7 at the time of writing, classes were instead tracked by the *gdb_objc_realized_classes* global variable. Enumeration of this variable, which is of type *NXMapTable*, allows recovery of the *objc_class* structure used to represent each class in memory. Inside of this structure are references to the definition of each class' variables and methods.

To find this per-class information, the data pointer of the class must be decoded. In Objective-C versions prior to macOS 10.15, this was accomplished by first following the *bits* member of the class. This value was then masked with the *FAST_DATA_MASK* value of *0x7ffffffffff8* for 64-bit systems. This then provided the address of the *class_rw_t* data structure, which provides references to the *class_ro_t* data structure along with other class values. *class_ro_t* can then be used to recover the base methods, name, instance variables, and properties of the class. Figure 2 shows the code snippet where we locate the *objc_class* using Volatility's pre-defined Object() method. The Object() method uses the address offset and the address space to locate a particular data structure. We then find the *bits* member and mask the value with the *FAST_DATA_MASK* which we have initialized earlier (not shown in the code). The offset thus obtained helps to locate the *class_rw_t* structure. In Figure 3, we see that the *class_rw_t* structure contains a data structure of type *class_ro_t*. If we look at the structure of the *class_ro_t*, we see the *name* field which can be used to recover the names of the classes.

Starting with macOS 10.15, the *class_rw_t* structure was changed so that the data pointer could point to either the *class_ro_t* data structure directly or to a *class_rw_ext_t* structure that contained the *ro* pointer. This change was encapsulated into a *PointerUnion* whose flag value determined which data structure the pointer referenced. This change breaks code that only understands the runtime before 10.15.

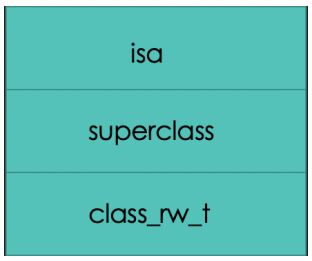

Figure 1: Objective-C Object

3.2. Locating Class Instances

The *objc_object* structure is used to represent each instance of a class. Figure 1 shows the layout of the *objc_object* data structure. To locate each instance of a particular

```
oclass = obj.Object("objc_class", offset = class_addr, vm = proc_as)
if not oclass.is_valid():
    return
bits_addr = oclass.bits.bits & FAST_DATA_MASK
bits = obj.Object("class_rw_t", offset = bits_addr, vm = proc_as)
if not bits.is_valid():
    return
name = proc_as.read(bits.ro.name, 256)
if not name:
    return
idx = name.find("\x00")
if idx != -1:
    name = name[:idx]
return name
```

Figure 2: Code Snippet for Finding Class Names

class, the heap region(s) of the process must be scanned. This occurs as Objective-C simply allocates the space for an object instance through *malloc* or *calloc* and does not specifically track each instance of afterwards.

Luckily, we can use the first member of *objc_object*, *isa*, as both a scanning marker as well as for context. This is possible as the *isa* variable points to the *objc_class* that that defines the object. By first enumerating all classes as described previously, we can then scan the heap memory and look for any addresses that reference a loaded class.

Since our recovery algorithm relies on scanning, our plugins not only find the current set of object instances, but also find de-allocated instances. This provides significant analysis capabilities as long-running processes can have a substantial number of de-allocated instances still left within the heap regions. Special care must be taken when parsing these instances though as pointers to variable data and method references can lead to buffers that have since been overwritten. This leads to a similar effect as memory-smearing [24], so our code validates each address and data type before processing it or using it as a result.

3.3. Parsing Instance Variables

Both instance variables and class methods are declared using the *entsize_list_tt* template. From the programmatic perspective, this template provides a generic list interface, but it is actually backed by an array. To implement this, the template stores the current count of elements, the size of each element, and a pointer to the beginning of the array of elements. This allows trivial enumeration in our plugins as Volatility has built-in support for arrays.

The set of instance variables for a class are stored as the *ivar_list_t* type, which uses the *entsize_list_tt* template. Each instance variable is tracked by an *ivar_t* structure,

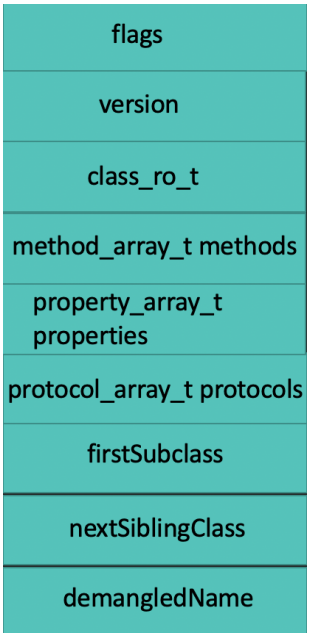

Figure 3: class_rw_t Data Structure

Figure 4: class_ro_t Data Structure

which stores the *offset*, *name*, and *type* of the variable. The offset specifies how many bytes away from the beginning of an *objc_object* instance is the value of a variable stored. The name and type are stored as strings. Figure 5 shows the Python code for locating the *ivar_list_t* data structure. We access each member of the *ivar_list_t* array using a for loop. The structure of the *ivar_t* data structure is shown in Figure 6. The code in Figure 5 shows how we read the name and type fields of the *ivar_t* data structure.

3.4. Decoding Variable Types

To accurately gather the value of a variable, the *type* string must be interpreted. These type representations are documented online [25], and our Volatility additions currently support parsing the encoded value of the following types:

- Single byte characters
- Signed and unsigned short integers
- Signed and unsigned four byte integers
- Signed and unsigned eight byte longs
- Floats and doubles
- Character (string) pointers
- Boolean values
- Class pointers
- Type pointers
- Arrays of supported types
- NSString, NSInteger, NSUrl, NSArray

Recovering the direct integer types is straightforward as they are simply stored as their little-endian value. Floats and doubles are stored such that the *f* and *d* type-specifiers to Python's *struct.unpack* can convert them. Pointers to classes are recursively parsed.

The NS types require separate parsing as they encode the data into separate, undocumented data structures. Binary analysis was required to understand how to recover variables of these types. Decoding variables of type *NSString* requires special care as it can be stored in numerous forms. In one encoding, the string is simply stored starting at the address of the data structure. In a second encoding, the string begins at offset sixteen (16) of the data structure, and in a third encoding, a pointer to a character string is stored at offset sixteen. Our code correctly recovers the target string in all three encodings that we uncovered during our analysis.

The *NSUrl* type was discovered to have a *NSString* that holds the URL at offset 0x18 of its data structure. *NSInteger* references the integer value directly. *NSArray*s store the size of the array at offset 4 and then the elements are listed contiguously after. The type of each element can be determined by following the *isa* pointer for each element. For non-class elements, such as *NSInteger*, the value can be directly interpreted.

3.5. Enumerating Class Methods

The set of methods for a class are stored as the *method_list_t* type, which also uses the *entsize_list_tt* template. Figure 8 shows how we access the method_list_t and locate each member of the array. Each method is represented by a *method_t* data structure. Figure 7 shows the layout of the *method_t* which tracks the method's name (sel), parameter specification (types), and implementation pointer (imp). As described in Section 6.3, our plugins automatically examine the code of each method to look for calls to often-abused methods and functions.

4. Swift Memory Analysis

4.1. Swift and Objective-C Interoperability

Swift supports two modes of operations. The first is standalone and, in this mode, the runtime does not rely on any Objective-C infrastructure nor does it interact with the Objective-C runtime. The second mode, which is used on all Apple platforms, enables interoperability between Objective-C and Swift. This allows programs written in the two languages to natively interact with each other and share APIs and data structures [26, 27].

Since the target of our research was Mac OS X systems, all samples that we analyzed had interoperability enabled.

```
ivar_list = obj.Object("entsize_list_tt<ivar_t, ivar_list_t, 0", offset = bits.ro.ivars, vm = proc_as)
if ivar_list.count>0 and ivar_list.count<1024:
    ivars = obj.Object(theType="Array", targetType="ivar_t", offset = ivar_list.first.v(), vm=proc_as, count=ivar_list.count)
    for ivar in ivars:
        ivar_name = proc_as.read(ivar.name, 32)
        if not ivar_name:
            continue
        idx = ivar_name.find("\x00")
        ivar_name = ivar_name[:idx]
        ivar_type = proc_as.read(ivar.type, 32)
        if not ivar_type:
            continue
        idx = ivar_type.find("\x00")
        ivar_type = ivar_type[:idx]
        yield ivar_name,ivar_type
```

Figure 5: Code Snippet for Finding Instance Variables

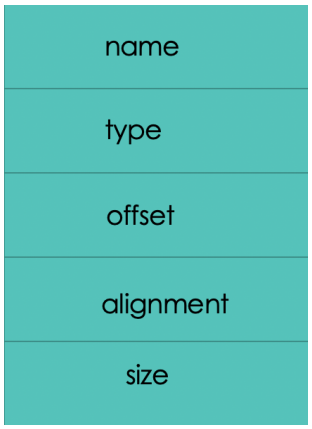

Figure 6: ivar_t Data Structure

Figure 7: method_t Data Structure

This allowed us to re-use some of our existing algorithms for enumerating data, but it also meant that we had to deeply study the Swift-specific parts of the runtime as well as the precise mechanisms that enable the interoperability.

4.2. Swift Metadata

A significant difference between Swift and Objective-C is that Swift encodes all information related to classes, variables, methods, and types into metadata records [28]. This required us to develop Volatility code that could parse the metadata records and derive the encoded values. It also required our plugins to be aware of when they are dealing with purely Objective-C classes versus a Swift class. These specifics are discussed in detail in the following sections.

Further adding to this complexity, the layout of the metadata structures has changed substantially between releases of Swift [29]. It was not until Swift 5.1, which was released in September 2019, that module stability was added [30]. From our understanding of reading related code and documents, this should prevent the addition of future metadata changes that are not backwards compatible.

4.3. Enumerating Loaded Classes

When runtime interoperability is enabled, Swift registers all loaded classes with the Objective-C runtime. This allows us to enumerate Objective-C and Swift classes using the same algorithm as described in Section 3.1 for Objective-C.

To determine if a loaded class is a Swift class, we check its data value address for either of the *FAST_IS_SWIFT_LEGACY* or *FAST_IS_SWIFT_STABLE* bits being set. These bits are set by the runtime when Swift classes are registered with the Objective-C runtime.

4.4. Locating Class Instances

As described in the Swift Type Metadata documentation [28], Swift class metadata records are also valid *objc_class* structures. When interoperability is enabled, the *isa* pointer serves the same purpose as in pure Objective-C, which is to reference the defining class of the instance.

```
methods_struct = obj.Object("entsize_list_tt<method_t, method_list_t, 3", offset = methods_struct.ptr.list, vm = proc_as)
if not methods_struct.is_valid():
    return
meth_count = methods_struct.count.v()
methods = obj.Object(theType="Array", targetType="method_t", offset = methods_struct.first.v(), count = meth_count, vm = proc_as)
method_names=[]
for meth in methods:
    method_name = proc_as.read(meth.name, 256)
    if not method_name:
        continue
    idx = method_name.find("\x00")
    method_name = method_name[:idx]
    method_names.append(method_name)
return method_names
```

Figure 8: Code snippet for finding methods

This allows us to follow the *isa* pointer as described in Section 3.2 to determine an instance's type.

To store the Swift-specific information, the Swift runtime a uses *objc_class* compatible metadata record named *TargetClassMetadata*. The beginning bytes of this record match the *obj_class* values and the Swift-specific information is added at the end.

4.5. Parsing Instance Variables

Because Swift class metadata records are also valid *objc_class* structures, our initial thought was to recover instance variable information using the technique described in Section 3.3. This approach did not work, however, because Swift does not propagate the type information for variables into their *ivar_t* structures, instead only propagating the name and offset fields.

To recover the types of the variables, which is required to properly parse instances of them, we had to instead rely on the Swift metadata record. Each *TargetClassMetadata* class holds a pointer to the *nominal type descriptor* for the class it represents. This descriptor is of type *TargetClassDescriptor*, and it holds information about the location of the methods, variables, superclass, storage size, and more of the target class.

By examining the class descriptor, we can determine the location of the metadata records used for reflection on the class. These records, of type *TargetTypeContextDescriptor*, hold the name and type of each variable of the class. By parsing these records, we can gather the type information that is purposely set to null in the *ivar_t* structures of Swift classes. This completes the needed set of type, name, and instance offset for each variable of a class.

We note that we could also recover the offset of each variable by parsing the *field offset vector* of the *TargetTypeContextDescriptor*, but this is unnecessary as we already gather it from the *ivar_t*. This would be required though to support Swift in non-interoperability mode on non-Apple platforms.

4.6. Decoding Variable Types

Unlike Objective-C, which stores the type string directly in the type member, Swift mangles all type names [31]. This prevented direct use of our existing Objective-C variable parsing code. Instead, we implemented a basic version of Swift's demangler to support the mostly common used types. This allows parsing the specific values most often encountered during our research.

We are currently investigating support for full name demangling in our plugins, but this will be a non-trivial task as the name mangling schema is quite complex and the format changed multiple times between Swift versions.

4.7. Enumerating Class Methods

Swift provides a number of mechanisms for classes to declare methods [32]. For the purposes of our research, we divide the many options into two categories, based on whether or not the method information is propagated to the Objective-C runtime. For methods that are propagated, we use the technique described in 3.5 to recover the class method information.

To recover methods that stay solely within the Swift runtime, we must start by inspecting the vtable information provided in the class metadata. This metadata is represented by a *TargetVTableDescriptorHeader* structure and stores the offset and size of the vtable. Inside the vtable is a *TargetMethodDescriptor* structure for every method of the class. Unfortunately, this method descriptor only stores the function pointer (*implementation address* in Swift parlance) of the described method.

In order to recover the method name and argument types, we then have to then cross-reference the function pointer with the symbol table of the analyzed executable. Volatility provides native support for parsing the symbols of Mach-O objects, which is the executable file format for Apple devices. Once a matching symbol is found, we then have the mangled representation, which encodes the function name and parameter types. We note that this is the same approach as described in [33].

By combining recovery of Objective-C methods and Swift methods, we are able to recover all methods of a class.

5. Malware Testbed Creation

To ensure that our algorithms were capable of detecting a wide variety of real-world malware, we gathered a diverse

set of samples that have been discovered in the wild and that were used in real attacks. Our research of which samples would be useful came from studying OS X malware analysis resources, such as Objective-See [34] and ESET reports [35], searches through threat intelligence sources, and through online searches of reported macOS malware infections and outbreaks.

For each malware sample, we documented its capabilities and the APIs, classes, and subsystems that it used to accomplish its malicious actions. This was accomplished through a mix of studying existing research as well as performing static and dynamic binary analysis. The results of this effort was a deep understanding of the system features that macOS malware abuses. We then used this knowledge to allow our plugins to generically detect this behaviour in running Objective-C and/or Swift applications.

We document our gathered malware dataset in the remainder of this section and then showcase our new plugins against them in the following section.

5.1. Crisis

As discussed in [21], Crisis was a highly sophisticated malware program used to spy on users, including recording of audio and web cameras, stealing browser data, and taking screenshots. It also deployed anti-forensics techniques to hide from live analysis.

5.2. EvilQuest

EvilQuest is a multipurpose malware that combines ransomware features along with data exfiltration and keylogging [36] [37].

5.2.1. FinSpy

FinSpy is a commercial surveillance tool that has been used against journalists and political activists throughout many countries in the world [38]. In late 2020, Amnesty International found a copy of this tool for macOS and published analysis of it [39] [40]. The results showed that the tool is capable of keylogging, recording of system audio, web camera recording, screen recording, sensitive file exfiltration, and more.

5.3. Komplex

Komplex is a trojan used by APT28, which is the Russian-backed APT group believed to be responsible for the 2016 Democratic National Committee hack as well as attacks against many other high-profile espionage targets [41]. Komplex provides the operator with information on running processes, the currently logged on user, and the victim system itself. It also allows the operator to write to any file on the file system or to execute any command.

5.4. MacDownloader

MacDownloader is the malware used by an APT group believed to be based out of Iran [42]. This malware gathers the list of running processes, installed applications, and saved browser data. It also prompts the user with a fake username and password dialogue to steal plain-text credentials. Finally, it gathers the victim's keychain files, which can be later decrypted using the stolen account credentials.

5.5. MacLoader

MacLoader is a trojan used by an APT group believed to be sponsored by North Korea. It has the ability to execute memory-only payloads that are delivered by an operator [43].

5.5.1. Realtime-Spy

Realtime-Spy is a commercial spyware suite that allows logging of all user activity, such as applications executed, keystrokes, emails, and websites visited, as well as capturing screenshots [44]. It came to notoriety when it was leveraged in a phishing campaign against users of the Exodus cryptocurrency software [44].

5.6. Ventir

Ventir is a Trojan found in the wild that provides operators with the ability to enable keylogging, downloading and uploading files, and commands execution [45] [46].

5.7. XAgentmacOS

XAgent is another trojan used by APT28, but it provides many more capabilities than Komplex [47]. Beyond the capabilities of Komplex, it also allows taking screenshots, stealing Firefox passwords, a remote shell, uploading files over FTP, and reading files from the file system.

6. Analysis Capabilities and Evaluation

Our goal in developing new Volatility plugins was to provide both automated detection of malware in a generic manner as well as empower investigators to add new capabilities specific to their investigations. In this section, we document our suite along with their algorithms.

6.1. Analyzing Loaded Classes

6.1.1. Plugin Algorithm

To detect the use of APIs and classes that are often abused for malicious purposes, we developed the *mac_analyze_classes* Volatility plugin. This plugin uses previously described algorithms to enumerate all Objective-C and Swift classes loaded in a process and then checks if any are in the known-bad list of classes. The plugin comes with a default list of known-bad classes as described in Table 2. This list can also be trivially extended by the analyst specifying a file name containing a list of classes with the plugin's *–class-list* option. This plugin provides an extremely

Name	Reason
popen	Allows running executables
NSTask::launch	Allows running executables
if_nametoindex	Used in gathering MAC addresses
NSCreateObjectFileImageFromMemory	Used for memory-only payload execution
CGEventTapCreate	Used for keylogging
CGEventTapEnable	Used for keylogging
IOHIDManagerRegisterInputValueCallback	Used for keylogging
CGWindowListCreateImage	Used in generating screenshots
CGContextClipToRect	Used in generating screenshots
UIGraphicsGetCurrentContext	Used in generating screenshots
method_exchangeImplementations	Used for method swizzling (hooking)
NSWorkspace:: runningApplications	Used to enumerate running processes

Table 1: List of functions and methods alerted on by default

powerful capability as malware must use runtime-provided APIs to access system hardware, gather system state, and to spawn other tasks, and the plugin successfully detects processes using such APIs.

6.1.2. Plugin Output

Figure 9 shows *mac_analyze_classes* against a sample infected with RealTime-Spy (RTS). As can be seen, the use of NSAppleScript by RTS is automatically detected without requiring hardcoding of RTS-specific indicators.

```
Process      Pid  Class
rts_sample   578  NSAppleScript
```

Figure 9: mac_analyze_classes detecting Realtime-Spy

6.2. Analyzing Instance Variables

6.2.1. Plugin Algorithm

The second plugin we developed was *mac_analyze_variables*. This plugin works by first using previously described algorithms to record the name, type, and offset of each instance variable of every loaded class in a process. It then finds each instance of every class and decodes the variables that are of supported instance types. By default, it will report the name and value of every instance variable of every class. This output can then be stored in a file and searched by the investigator for patterns of interest and/or indicators of compromise. The ability to automatically find all strings, which can include file paths, URLs, command and control configuration data, and more, is a powerful capability that the plugin provides.

Besides simply extracting all variables, the plugin also supports two modes of operation. The first is through *–patterns*, which specifies the path of file containing a list of search terms of interest. This can be used to filter output to only variables pertaining to a particular investigation or set of IOCs. The second is through *–variables*, which allows investigators to specify comma-separated lists identifying class names and members names of interest. The plugin will then only report the presence and decoded contents of variables matching the list. This allows IOCs of known-bad classes and variables to be automatically found and reported in samples. Such a capability is particularly useful when an investigator wants to scan many samples in parallel to determine which, if any, are infected. Given the size and scope of modern investigations, these scalable approaches are required.

6.2.2. Plugin Output

While analyzing the Ventir malware, we noticed that there was a variable of type *NSString* named *path_res* inside of its *UITimer* class. This variable caught our attention as *mac_analyze_variables* showed that it contained a path inside of our lab user account's home folder. We investigated further and saw this this path was the location in which the malware stores its executables and configuration data. Having this information, we could then make a simple comma separated file containing "UITimer, path_res" and use this file's path as the argument to *–variables*. Figure 10 shows the output of the plugin when ran against a Ventir infected memory sample and our given configuration file. As can be seen, the decoded value is automatically reported as well as the malicious process' information.

```
Process  Pid    Class    Variable  Value
Ventir   11538  UITimer  path_res  /Users/bob/Library/.local
```

Figure 10: mac_analyze_variables detecting Ventir

6.3. Analyzing Functions and Class Methods

Beyond analyzing just the loaded classes and the current values of the instance variables, we also wanted to provide deep analysis of the code running inside of Objective-C and Swift processes. During investigation of our mal-

Name	Reason
NSPasteboard	Allows clipboard monitoring
AVAudioRecorder	Used to access microphone audio
AVCaptureDeviceInput	Used to monitor web cameras
NSEvent	Used to monitor the keyboard, mouse, and other devices
NSAppleScript	Allows running scripts and gathering data

Table 2: List of classes alerted on by default

ware analysis dataset, we realized that this meant we needed to analyze both the methods associated with each class as well as the globally accessible methods not associated with a particular class. To meet this criteria, we developed the *mac_analyze_code* plugin, which provides automated analysis of code inside of Objective-C and Swift processes and reports usage of functions that are known to be abused by malware.

6.4. Suspicious APIs

During analysis of our malware dataset, we discovered many runtime-provided APIs that allow for a wide range of system abuse. Most of these were methods not associated with a particular Objective-C or Swift class, but instead are lower level APIs also available to code in other programming languages, such as C and C++. We did however find several methods of Objective-C classes that allowed for abuse. To avoid creating false positives in *mac_analyze_classes* by alerting on any usage of these classes, we instead incorporated detecting only uses of the methods that are known to be abused. Table 1 lists the methods that *mac_analyze_code* considers suspicious and automatically alerts to when found. As with the other plugins, this list can trivially be extended by analysts through text files.

```
Process   Pid   Class                      Method                 Suspicious
macloader 1596  Authentication Controller  getRunningProcessList  NSWorkSpace::runningApplications
```

Figure 11: mac_analyze_code detecting Mac Loader

6.4.1. Enumerating Functions to be Analyzed

mac_analyze_classes finds the functions and methods to analyze through two sources. First, the methods of all loaded Objective-C and Swift classes are enumerated using the previously described algorithms. Next, the *implementation address* pointer for each method is recorded. The second source used is the symbol table of the executable being analyzed. This will contain the address of all functions, not just those that are methods of a class. This was determined to be necessary as we found several instances of malware that used global functions as part of its malicious codeflow.

6.4.2. Code Analysis Algorithm

The plugin analyzes code through use of the distorm [48] disassembly library. This library provides the plugin with parsable instructions for each instruction of a function and any sub-functions that are also called. When the plugin encounters a *CALL* instruction, which is executed to call a function in a program, it compares the address being called to the addresses of known suspicious functions. The mapping of known suspicious functions to their runtime addresses in a particular process is built by enumerating the symbol table of all libraries mapped into the process. Through this comparison, the plugin is able to report which processes are hosting code that calls suspicious functions as well as context of the calling function.

6.4.3. Plugin Output

Figure 11 shows the output of *mac_analyze_classes* against a memory sample infected with MacLoader. In the output, it can be seen that the *getRunningProcessList* method of the malware's *AuthenticationController* class calls the suspicious function *NSWorkspace::runningApplications*. This immediately tells the investigator of the process hosting potentially malicious code as well as where to begin any reverse engineering efforts.

7. Conclusions

In this paper, we have presented novel memory forensic methods for automated analysis of modern Objective-C and Swift runtimes. The plugins developed as a part of this effort allow for investigators of all skill levels to detect malware and other suspicious code running on a system in a straightforward manner. These plugins also allow for automated extraction of variable data of applications, which can assist in investigations beyond malware, such as insider threats. The plugins are all easily extendable through basic text files that allow for filtering of output and incorporation of IOC data. Given the popularity of Apple laptops and desktops in the corporate environment as well as the continued rise in home environments, deeper inspection of macOS userland runtimes is now a required feature of memory analysis frameworks. We have developed this capability for the Volatility framework, and our goal is to have these capabilities integrated directly into the framework after publication of this paper. This will allow the capabilities to be usable by the entire digital forensics and incident response community.

References

[1] B. Vogel, More than Half of All Companies Use Mac - Parallels Survey, https://www.parallels.com/blogs/mac-survey, 2020.

[2] D. Palmer, Hackers are targeting MacOS users with this updated malware, https://www.zdnet.com/article/hackers-are-targeting-macos-users-with-this-updated-malware/, 2020.

[3] Kaspersky, New MacOS X backdoor variant used in APT attack, https://securelist.com/new-macos-x-backdoor-variant-used-in-apt-attacks/33214, 2012.

[4] J. Long, Operation AppleJeus and OSX/Lazarus: Rise of a Mac APT, https://www.intego.com/mac-security-blog/operation-applejeus-and-osxlazarus-rise-of-a-mac-apt, 2018.

[5] L. O'Donnell, MacOS Users Targeted By OceanLotus Backdoor, https://threatpost.com/macos-users-targeted-oceanlotus-backdoor/161655, 2020.

[6] P. Stokes, Four Distinct Families of Lazarus Malware Target Apple's macOS Platform, https://www.sentinelone.com/blog/four-distinct-families-of-lazarus-malware-target-apples-macos-platform, 2020.

[7] A. Tiberius, B. Botezatu, Dissecting the APT28 Mac OS X Payload, http://download.bitdefender.com/resources/files/News/CaseStudies/study/143/Bitdefender-Whitepaper-APT-Mac-A4-en-EN-web.pdf, 2017.

[8] T. V. Foundation, The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework, https://github.com/volatilityfoundation/volatility, 2016.

[9] DFRWS, DFRWS Online, http://old.dfrws.org/2005/challenge, 2005.

[10] G. Garner, Knt tools, http://www.gmgsystemsinc/knttools, 2005.

[11] N. Petroni, A. Walters, T. Fraser, W. Arbaugh, Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory, Digital Investigation 3 (2006). 2006.

[12] A. Arasteh, The vad tree: a process-eye view of physical memory, in: Proceedings of the 2007 Digital Forensic Research Workshop, pp. 62–64. 2007.

[13] B. Dolan-Gavitt, Forensic analysis of the Windows registry in memory, https://www.sciencedirect.com/science/article/pii/S1742287608000297, 2008.

[14] R. v. Baar, W. Alink, A. R. v. Ballegooij, Forensic memory analysis: Files mapped in memory, https://www.sciencedirect.com/science/article/pii/S1742287608000327, 2008.

[15] DFRWS, DFRWS Online, http://old.dfrws.org/2008/challenge, 2008.

[16] Volatility, Volatility 2.3 Released! (Official Mac OS X and Android Support), https://volatility-labs.blogspot.com/2013/10/volatility-23-released-official-mac-os.html, 2013.

[17] A. Case, Forensic memory analysis of android's dalvik vm, http://dfir.org/research/android-memory-analysis.pdf, 2011.

[18] H. Macht, Dalvikvm support for volatility, http://lists.volatilesystems.com/pipermail/vol-dev/2012-October/000187.html, 2012.

[19] A. Ali-Gombe, S. Sudhakaran, A. Case, G. G. Richard III, DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction, in: RAID, pp. 547 – 559. 2019.

[20] M. Ligh, Omfw 2012: Malware in the windows gui subsystem, https://volatility-labs.blogspot.com/2012/10/omfw-2012-malware-in-windows-gui.html, 2012.

[21] A. Case, G. G. Richard III, Detecting objective-c malware through memory forensics, Proceedings of the 16th Annual Digital Forensics Research Workshop (DFRWS 2016) (2016). 2016.

[22] L. Myers, New Apple Mac Trojan Called OSX/Crisis Discovered, https://www.intego.com/mac-security-blog/new-apple-mac-trojan-called-osxcrisis-discovered-by-intego-virus-team, 2016.

[23] Apple, Programming with Objective-C, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, 2014.

[24] H. Carvey, Page smear, http://seclists.org/incidents/2005/Jun/22, 2005.

[25] Mattt, Type Encodings, https://nshipster.com/type-encodings/, 2020.

[26] D. Ramirez, Understanding objective-c and swift interoperability, https://rderik.com/blog/understanding-objective-c-and-swift-interoperability/, 2021.

[27] mikeash, Objective-c and swift interop, https://github.com/apple/swift/blob/a73a8087968f9111149073107c5242d83635107a/docs/ObjCInterop.md, 2020.

[28] Apple, apple/swift, https://github.com/apple/swift/blob/main/docs/ABI/TypeMetadata.rst, 2020.

[29] Swift 5 Type Metadata Detailed, https://www.programmersought.com/article/4671623102/, 2018.

[30] Apple, Swift 5.1 Released!, https://swift.org/blog/swift-5-1-released/, 2019.

[31] S. Documentation, Mangling, https://github.com/apple/swift/blob/main/docs/ABI/Mangling.rst, 2021.

[32] Apple, Swift.org, https://docs.swift.org/swift-book/LanguageGuide/Methods.html/, 2020.

[33] D. Selander, Building a class-dump in 2020, https://derekselander.github.io/dsdump/\#swift_methods_in_a_class, 2020.

[34] P. Wardle, See, https://objective-see.com/, 2018.

[35] ESET, https://www.eset.com/us/, 1987.

[36] P. Stokes, EvilQuest Rolls Ransomware, Spyware & Data Theft Into One, https://www.sentinelone.com/blog/evilquest-a-new-macos-malware-rolls-ransomware-spyware-and-data-theft-into-one, 2020.

[37] P. Wardle, OSX.EvilQuest Uncovered, https://objective-see.com/blog/blog_0x59.html, 2020.

[38] A. International, Phishing attacks using third-party applications against Egyptian civil society organizations, https://www.amnesty.org/en/latest/research/2019/03/phishing-attacks-using-third-party-applications-against-egyptian-civil-society-organizations, 2019.

[39] A. International, German-made FinSpy spyware found in Egypt, and Mac and Linux versions revealed, https://www.amnesty.org/en/latest/research/2020/09/german-made-finspy-spyware-found-in-egypt-and-mac-and-linux-versions-revealed/, 2020.

[40] P. Wardle, FinFisher Filleted, https://objective-see.com/blog/blog_0x4F.htm, 2020.

[41] D. Creus, T. Halfpop, R. Falcone, Sofacy's 'Komplex' OS X Trojan, https://unit42.paloaltonetworks.com/unit42-sofacys-komplex-os-x-trojan/, 2016.

[42] C. Guarnieri, C. Anderson, iKittens: Iranian Actor Resurfaces with Malware for Mac (MacDownloader), https://iranthreats.github.io/resources/macdownloader-macos-malware/, 2017.

[43] D. Goodin, Newly discovered Mac malware uses "fileless" technique to remain stealthy, https://arstechnica.com/information-technology/2019/12/north-koreas-lazarus-hackers-up-their-game-with-fileless-mac-malware/, 2019.

[44] J. Long, Privacy Exodus: spam delivers Mac spyware, https://www.intego.com/mac-security-blog/privacy-exodus-spam-delivers-mac-spyware, 2018.

[45] D. Erwin, 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, 2014.

[46] M. Kuzin, The Ventir Trojan: assemble your MacOS spy, https://securelist.com/the-ventir-trojan-assemble-your-macos-spy/67267/, 2014.

[47] R. Falcone, XAgentOSX: Sofacy's XAgent macOS Tool, https://unit42.paloaltonetworks.com/unit42-xagentosx-sofacys-xagent-macos-tool/, 2017.

[48] Gdabah, gdabah/distorm, https://github.com/gdabah/distorm, 2020.