Advancing Memory Forensics Techniques for Apple M1 Architecture

Author: Sunny Thakur

Table of Contents

Abstract

The landscape of malware threats is advancing, with attackers leveraging increasingly sophisticated tactics. By exploiting comprehensive application APIs—such as Objective-C's—attackers can collect user activity data, initiate background processes covertly, and conduct other malicious actions. For certain malware, like those executing only in memory, memory forensics becomes essential for retrieving evidence of such activities. The introduction of Rosetta 2 on Apple's M1 chip has added a new attack vector, enabling binaries for both Intel x86_64 and ARM64 architectures to run in user mode. Consequently, it is crucial that forensic tools quickly adapt to recognize indicators of malicious behavior in memory samples across emerging platforms. This paper provides a memory analysis of the Rosetta 2 runtime using the Volatility Framework, offers curated memory samples to validate Volatility plugins and algorithms, and discusses enhancements to support Apple M1 in Volatility's forensic capabilities.

Chapter 1: Introduction

This research aims to enhance memory forensics capabilities on Apple's new M1 ARM-based chip, addressing its unique architecture and expanding existing memory analysis frameworks.

1.1 Memory Forensics

Memory forensics, a critical aspect of digital forensics, enables in-depth analysis of volatile memory (RAM) during incident responses or investigative research. It offers invaluable insights into a system's state at the time of memory capture, allowing forensic experts to recover artifacts like active and recently terminated processes, open files, registry information, network connections, and more. For scenarios involving memory-only malware or encrypted data, memory forensics can be the sole means to investigate a system comprehensively.

To analyze memory data, tools are needed to parse the acquired sample and extract valuable information. The Volatility Framework is widely used in the cybersecurity field for memory sample analysis. With the growing sophistication of malware, it’s essential that Volatility remains compatible with new hardware platforms. This research specifically seeks to extend Volatility’s functionality to Apple’s M1 chip by generating M1-compatible memory samples to enable the updating of existing plugins and development of new ones.

A prime example of the critical role of memory forensics can be seen in the 2015 Duqu 2.0 discovery. Kaspersky Labs identified this sophisticated malware, which operates entirely in-memory, making it challenging to detect without memory analysis. Similarly, Cobalt Strike, commonly used by attackers, operates in-memory, underscoring the importance of memory forensics in countering modern threats.

1.2 The Apple M1 Chip

In November 2020, Apple launched its ARM64-based M1 chip, marking a significant shift from Intel’s x86-64 architecture. The M1 system-on-a-chip (SoC) design integrates components like a unified memory architecture, delivering notable performance boosts. With subsequent releases like the M1 Pro, M1 Max, and M1 Ultra, Apple has continued enhancing this architecture with higher core counts, more powerful GPUs, and increased memory capacity, pushing the boundaries of CPU and GPU performance.

However, this shift from Intel’s x86-64 to ARM64 has introduced software compatibility challenges. To bridge the gap, Apple released Rosetta 2, a runtime environment that translates x86 binaries for ARM, similar to the original Rosetta layer used during the 2006 transition from PowerPC to Intel. Rosetta 2 launches when an x86 binary is run on an M1 device, translating the binary at runtime, which this research explores further in Chapter 3.

1.3 Importance of This Research

Examining Rosetta 2 from a memory forensics standpoint is essential for several reasons. First, it introduces an additional attack vector, as existing x86 malware can potentially execute on M1 devices without modification. Second, memory forensics has traditionally focused on kernel-level malware, leaving userland—where malware can operate more easily—under-examined. As macOS continues to limit kernel access, analyzing userland activities becomes critical for uncovering malware operations.

Additionally, memory forensics remains pivotal in real-world incident response, safeguarding both user privacy and organizational security. This research assesses current forensic tools for detecting malware on M1, identifies areas for enhancement, and creates a collection of memory samples featuring M1-specific artifacts, essential for validating new and updated plugins.

1.4 Outline

This paper is organized as follows:

- **Chapter 2**: Covers background information on the Volatility Framework, current M1 memory forensics, userland malware analysis, and Rosetta 2.
- **Chapter 3**: Details the environment setup and memory sample acquisition from M1 systems.
- **Chapter 4**: Describes API testing on Rosetta 2, including findings and analysis.
- **Chapter 5**: Summarizes parallel contributions.
- **Chapter 6**: Concludes the research and outlines future directions.

Chapter 2: Background and Related Work

In this chapter, we provide context on related works, introducing essential concepts and past research relevant to this study.

2.1 The Volatility Framework

The Volatility Framework is an open-source toolset implemented in Python, widely used for memory analysis in incident response and malware investigation. Key reference material for this framework is *The Art of Memory Forensics* by Leigh et al., which provides comprehensive background on system internals and plugin functionality for Windows, Linux, and macOS.

One of Volatility's strengths lies in its extensibility through plugins, each focusing on specific data extractions from memory, such as processes, network connections, or encrypted data. Developing reliable plugins requires validated memory samples with known artifacts to ensure plugins extract the intended information. This study seeks to create memory samples containing M1-specific data to aid in developing and testing plugins compatible with Apple Silicon.

2.2 Memory Forensics on Intel macOS

Historically, memory forensics has focused on kernel-level malware, as control over the kernel allows broad system manipulation. However, as macOS enhances kernel security, userland has become a more attractive target for malware. Case et al. (2016) and Manna et al. (2019) contributed significantly to macOS userland forensics by developing Volatility plugins for Objective-C method analysis, a foundation for selecting Objective-C functions to validate in this study.

2.3 Analyzing the M1 Chip

Since the M1's release, few detailed technical studies have been published due to its novelty. Prior work, including Duke's 2021 analysis of standard macOS Volatility plugins on the M1, highlighted architectural differences affecting plugin functionality, aiding Volatility developers in refining plugins for the M1.

2.4 M1 Research Efforts

Related research includes Corellium's 2021 BlackHat presentation on reverse-engineering the M1 and ZecOps' Objective By the Sea presentation on M1 exploitation techniques. These studies provide valuable technical insights into the M1's unique architecture, which informed aspects of this research.

2.5 Rosetta 2

Rosetta 2 enables Intel-based apps to run on M1, bridging the gap during Apple's transition to ARM64. This translation layer supports two methods of x86 execution: through Universal 2 binaries (dual architecture Mach-O files) or Rosetta 2. Rosetta dynamically translates x86 apps into an ARM-compatible format, excluding kernel extensions and certain virtualization applications. Nakagawa's Project Champollion, a significant Rosetta 2 study, explored its architecture as a potential attack vector, contributing patches to Ghidra for better compatibility.

Chapter 3: Memory Acquisition and Environment Setup

In this chapter, we will detail the tools utilized and the configuration process essential for this research.

3.1 Memory Acquisition

Acquiring a memory sample from the target system posed a challenge in the setup phase. For Intel-based systems, two primary methods exist for memory acquisition: virtualization software, such as VMware or Parallels, and software-based acquisition tools installed on the host system. Virtualization software typically supports snapshotting the guest operating system, capturing memory contents for analysis. Alternatively, software-based acquisition tools on the host can directly dump visible physical memory.

However, neither VMware nor Parallels could meet the requirements for this research. Memory analysis on a virtual machine (VM) depends on the ability to suspend the VM and take snapshots. Currently, these features are unavailable for the M1 chip. VMware Fusion is still in beta for the M1, with stable releases expected later, while Parallels, though stable for M1, lacks the snapshotting functionality needed for memory acquisitionn, we opted for a software-based acquisition tool, Surge Collect Pro, licensed temporarily by Volexity. Surge Collect Pro enabled us to capture memory samples directly from the host M1 system without relying on VMs. Our sincere thanks to Volexity for their support, as this research would not have been feasible without the ability to obtain native M1 memory samples.

3.2 Environment Setup

The primary system used in this study was a 2021 Apple 24-inch iMac equipped with the M1 chip, running macOS 12.1 Monterey, 16GB of RAM, and 1TB of storage. Code development and testing were performed on a 2019 MacBook Pro with an Intel Core i9, also running macOS 12.1 Monterey. Objective-C binaries, discussed in the following chapter, were developed and tested using XCode v13.2.1 on this Intel MacBook Pro.

To utilize Surge Collect Pro, we installed a custom kernel extension (kext) on the M1 machine. Kernel extensions provide the elevated privileges required for low-level tasks in kernelspace, and in this case, the kext allowed Surge Collect Pro access to physical memory. macOS 11 and later require booting into Recovery Mode, setting the security level to Reduced Security, and granting permissions under Security and Privacy in System Settings to load third-party kexts . Once configured, the commality provided by Surge enabled efficient memory acquisition from the system.

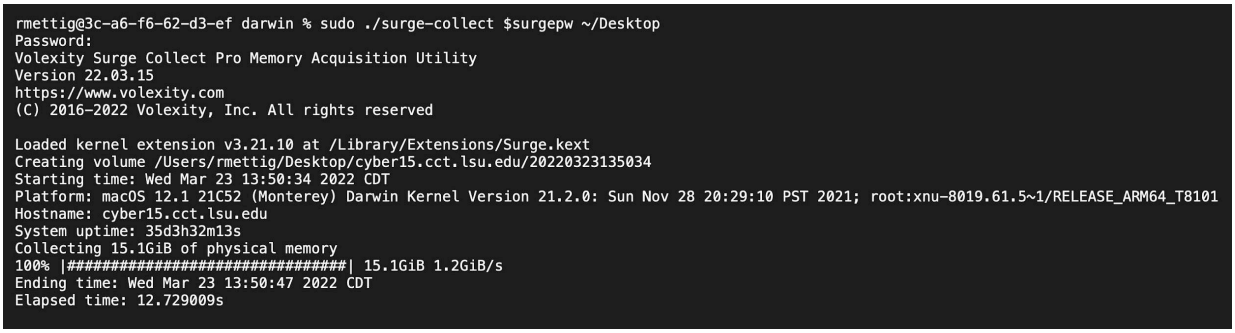


Figure 3.1. Figure shows Surge Collect Pro getting a memory sample from the M1 host

To accommodate Intel binaries, we installed Rosetta 2, which is not pre-installed with macOS. Rosetta 2's installation prompt appears the first time an Intel binary is executed.

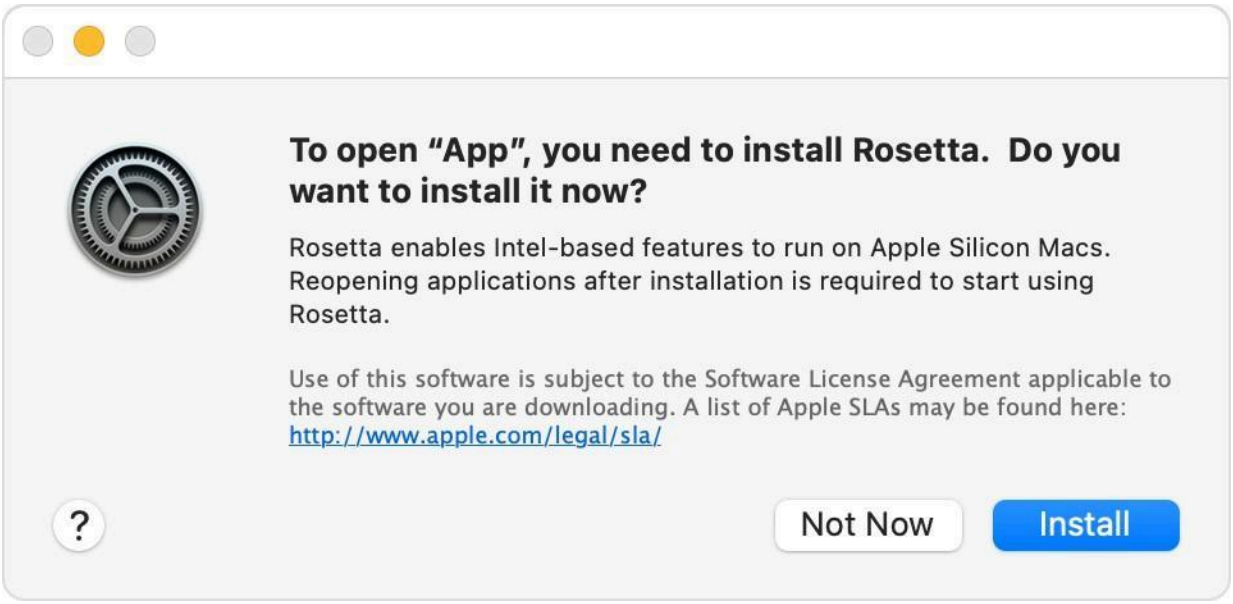


Figure 3.2. Rosetta 2 installation prompt (Source: https://support.apple.com/enus/HT211861)

Finally, we disabled System Integrity Protection (SIP) to access the oah folder containing the Ahead-of-Time (AOT) translation files. SIP restricts system directories to processes signed by Apple, even from root users . Disabling SIP required booting into ode, entering csrutil disable in the Terminal, and then rebooting the system.

3.2.1 Volatility for M1

For this project, we utilized an internal version of the Volatility Framework, v2.6.1, modified to support ARM64 and M1-based analysis. M1 kernel profiles were developed to allow Volatility to interpret kernel states at the time of memory acquisition. These profiles contain symbol and type information for a specific kernel version and were generously provided by Volexity.

In addition, configuring Volatility required an ARM64-compatible address space. Address spaces facilitate virtual-to-physical memory address translation, essential for parsing memory samples. Tsahi Zidenberg contributed an ARM64 address space to his Volatility fork, which served as the foundation for address translation on M1 systems .

When analyzing macOS memory samples with Vol is also necessary to provide the Address Space Layout Randomization (ASLR) shift and the kernel page tables' physical offset (DTB), which change with each reboot. Fortunately, this data is included in a JSON file generated by Surge following memory acquisition.

Further additions and modifications to Volatility and the memory analysis process will be discussed in later chapters.

Chapter 4. Memory Analysis

The chapter on **Memory Analysis** provides a detailed breakdown of methods used to examine and identify malicious behavior in macOS, specifically focusing on testing Objective-C API functions commonly exploited by malware and investigating Rosetta 2 for new memory analysis artifacts. This approach includes creating a set of memory samples to verify and support Volatility plugins for macOS, with an emphasis on compatibility with the Apple M1 chip.

4.1 Testing the Objective-C API

To gather memory samples, several Objective-C and C++ functions were tested, selected based on known abuse cases reported in macOS malware studies. These functions are critical for various attack vectors, including keylogging, launching executables, memory-only execution, and clipboard monitoring.

Table 4.1. Selected Objective-C API Functions/Classes for Testing

Name	Reason
addGlobalMonitorForEventsMatchingMask	Keylogging
NSEvent	Monitoring devices (keyboard, mouse, etc)
CGEventTapCreate	Keylogging
CGEventTapEnable	Keylogging
NSTask::launch	Allows running executables
NSAppleScript	Run scripts and data gathering
NSCreateObjectFileImageFromMemory	Memory-only execution
NSPasteboard	Clipboard monitoring

This list is by no means comprehensive and may be expanded at the analyst's discretion, but it does cover different basic yet relevant use cases that have been reported . All of the binaries were run one-at-a-time and individual memory samples were collected for each. The main idea is to make sure the programs load the classes/objects to memory so that they can be examined and extracted later. In between each sample, permissions for the Terminal in System Preferences were reset on the M1 to check whether any system prompts would pop up. This will be particularly relevant in the case for the keyloggers. The following examples are grouped by category and are not presented in any particular
order.

4.1.1. Keylogging

Keyloggers are a type of spyware, a class of software designed with the purpose of tracking a user's activity without their knowledge, that specifically aim to log all of a user's input to a device, usually a keyboard or mouse. While the keylogger runs in the background, it may write to a file that gets stored in a potentially inconspicuous location on the device and can be retrieved later by a malicious actor. Memory forensics is particularly powerful against malware of this nature as it is able to uncover any loaded classes as well as file descriptors related to each process.

One interesting finding is that, for both cases that will be discussed, a prompt popped up asking for Accessibility Permissions to be granted to the Terminal in order to run the programs.

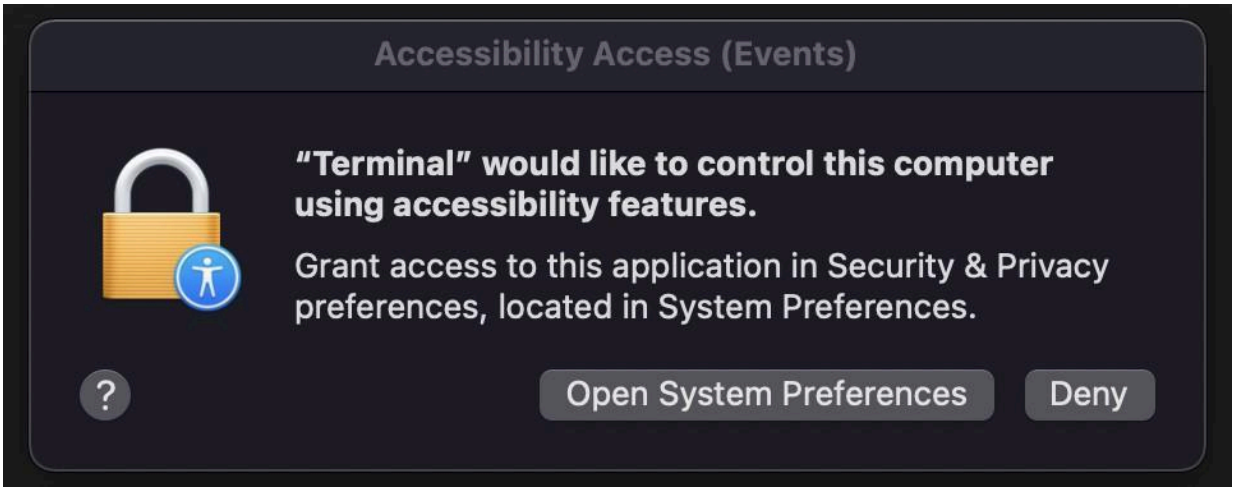


Figure 4.1. Terminal prompt requesting Accessibility Permissions.

Since the permissions were reset between running each keylogger, both triggered this notification. However, if the Terminal app already has permission prior to running the executable then it is possible to run the program without the user's knowledge.

Four of the selected items from the list above were grouped into two programs and tested for their keylogging functionality.

- **NSEvent / addGlobalEventMonitor**

NSEvent is a class that is a part of Apple's AppKit framework, which provides multiple objects for event-driven interactions with user interfaces. One of those objects is addGlobalEventMonitor which creates an event object that is capable of monitoring user keystrokes outside the main keylogger process.

Written in Objective-c, this simple keylogger doesn't output to a file. It mainly uses the event monitor to redirect the input – which can already be seen printed in the terminal – back to the terminal one at a time.

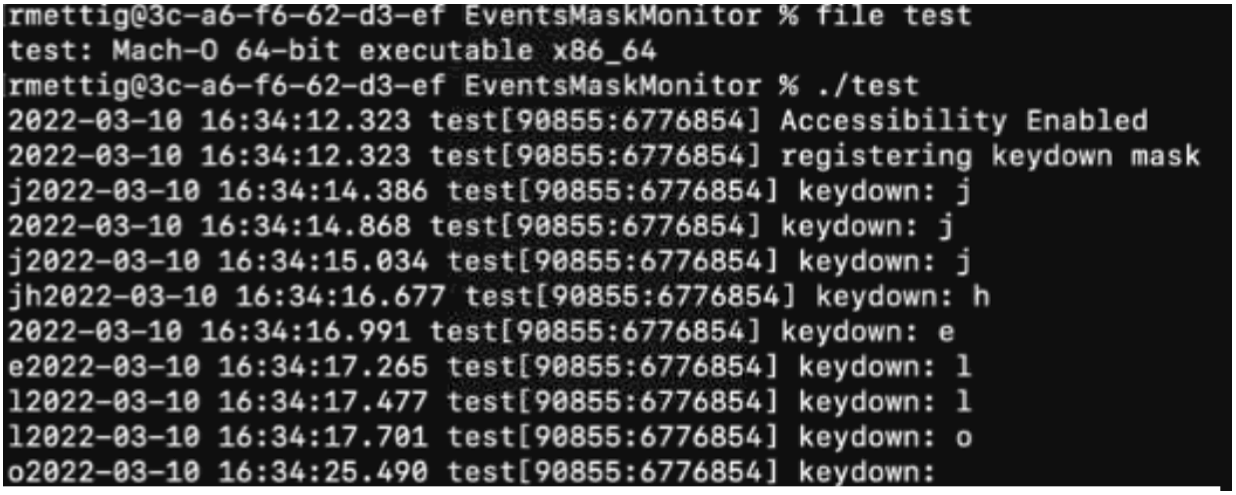


Figure 4.2. Volatility - global event handler running on the command line

One thing that was added to this example was a check for the Accessibility Permissions. It was interesting to note that, if the permissions for the Terminal were disabled, the program would still run but the event monitor was not able to capture the input. In the end the only memory sample taken was that of the event monitor running with accessibility permissions enabled, but it could potentially be interesting to do a comparison of the executable with permissions disabled.

While it has been previously possible to enumerate all Objective-C classes in memory on an Intel machine, this capability is still not available for the M1 and only limited analysis of that nature can be done here. However, we can still verify that the program can be found in memory as well and have it dumped for further manual analysis.

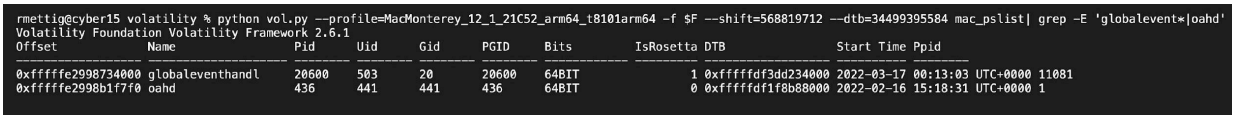


Figure 4.3. Volatility output showing the global handler keylogger and oahd running in memory in the process list

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F --shift=568819712 --dtb=34499395584 mac_procdump -D out --pid=20600
Volatility Foundation Volatility Framework 2.6.1
Task                    Pid    Address           Path
----------------------- ------ ----------------- ----
globaleventhandl          20600 0x000000010285c000 out/task.20600.0x10285c000.intel.dmp
```

Figure 4.4. Volatility output showing the global event-handler binary dumped form memory

- **CGEventTapCreate / CGEventTapEnable**

CGEventTapCreate and CGEventTapEnable are both methods from Apple CoreGraphics framework. tapCreate allows for an event tap to be created, and if successful, tapEnable enables that tap to be accessed. The taps allow access to events when a user manipulates an input device such as a keyboard or mouse.

For this example, we used a proof of concept keylogger written by Casey Scarborough [27] that is written in C and that was able to compile and run after a few minor modifications. This specific example will actually output to a file that will be stored on disk. One of the modifications necessary was the location of the logfile because the original path was at /var/log/keystroke.log, and /var is one of the directories protected by SIP. At the time this keylogger was tested, SIP was still enabled on the M1 and would not let the process make any modifications to the logfile, so the program would error out. However, after changing the logfile location, the program was able to run without further problems.

The next step was to collect the memory sample and verify that we can find the original binary in memory as well as the log file. As you can see, Volatility sucessfully detected the

```
rmettig@cyber15 keylogger-clone % ./keylogger-x86_64
ERROR: Unable to open log file. Ensure that you have the proper permissions.
rmettig@cyber15 keylogger-clone %
```

Figure 4.5. Error thrown when the program tried to open the logfile created under SIP protected path.

```
[rmettig@cyber15 keylogger-clone % ./keylogger
 Logging to: /Users/rmettig/Desktop/keystroke.log
 hjkasd
 hello my name is raphaela
 this is loggin me
 ^C
[rmettig@cyber15 keylogger-clone % ./keylogger
 Logging to: /Users/rmettig/Desktop/keystroke.log
 ok keylogging still works then
 hello world
 ^C
[rmettig@cyber15 keylogger-clone % vim Makefile
 rmettig@cyber15 keylogger-clone %
```

Figure 4.6. CGEventTap Keylogger Running

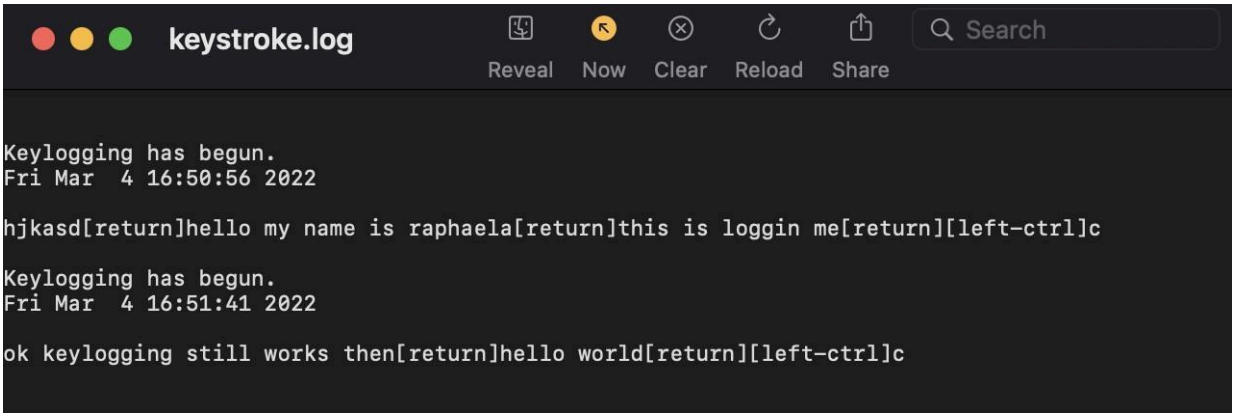
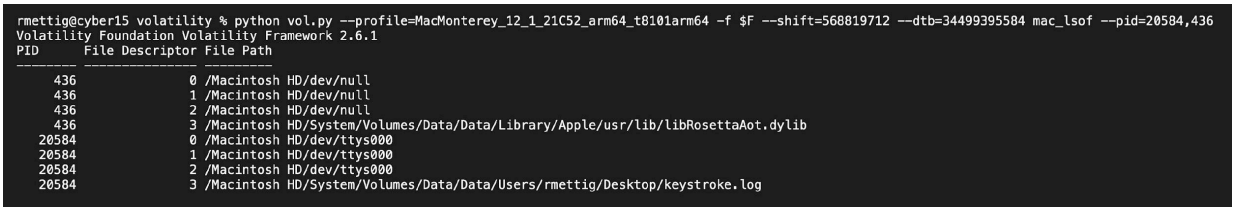


Figure 4.7. CGEventTap Keylogger Logfile

running executables and their file descriptors pointing to the logfile in memory, as well as the Rosetta dynamic library.

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F --shift=568819712 --dtb=34499395584 mac_pslist | grep -E 'keylogger|oahd'
Volatility Foundation Volatility Framework 2.6.1
Offset             Name                 Pid      Uid      Gid      PGID     Bits         IsRosetta DTB               Start Time Ppid
------------------ -------------------- -------- -------- -------- -------- ------------ --------- ----------------- ---------- --------
[snip]
0xfffffe299875dbf8 keylogger            20584    503      20       20584    64BIT                1 0xfffffdf0e98e8000 2022-03-17 00:12:18 UTC+0000 11081
0xfffffe2998b1f7f0 oahd                 436      441      441      436      64BIT                0 0xfffffdf1f8b88000 2022-02-16 15:18:31 UTC+0000 1
```

Figure 4.8. Volatility output showing the CGEventTap keylogger and oahd running in memory in the process list



```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F --shift=568819712 --dtb=34499395584 mac_lsof --pid=20584,436
Volatility Foundation Volatility Framework 2.6.1
PID      File Descriptor File Path
-------- --------------- ---------
     436               0 /Macintosh HD/dev/null
     436               1 /Macintosh HD/dev/null
     436               2 /Macintosh HD/dev/null
     436               3 /Macintosh HD/System/Volumes/Data/Data/Library/Apple/usr/lib/libRosettaAot.dylib
   20584               0 /Macintosh HD/dev/ttys000
   20584               1 /Macintosh HD/dev/ttys000
   20584               2 /Macintosh HD/dev/ttys000
   20584               3 /Macintosh HD/System/Volumes/Data/Data/Users/rmettig/Desktop/keystroke.log
```

Figure 4.9. Volatility output showing open file handles used by CGEventTap keylogger and oahd in memory

4.1.2. Launching Executables

Launching executables is another important vector that we need to pay attention to. It is a common technique for malware to use downloaders as an initial infection vector, which checks their running environment for information before doing anything else. An example of this is Proton.B malware for macOS X, which uses NSTask Launch (discussed next) to launch a bash process to check whether its persistence mechanism has been installed and executed [25]. A common goal for malicious software is to launch a subprocess that has some degree of control over the system and can easily return system-related information.

- **NSTask::launch**

NSTask is a class from Apple's Foundation Framework, which provides access to system-related objects and tools. In this case, an NSTask object represents a subprocess of the current process. By providing it with a path to the target subprocess and the required arguments, it is fairly straightforward to run and control the desired program.

For this example, you can see the output of the mac_pstree plugin, which lists all the parent-child process trees in the memory dump being analyzed.

The program I wrote uses NSTask to launch /bin/sleep, which can be seen as a child process for nstask_updated_sleep.

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F
--shift=568819712 --dtb=34499395584 mac_pstree
Volatility Foundation Volatility Framework 2.6.1
kernel_task          0               0
.launchd             1               0
[snip]
..Terminal           11079           503
...login             75101           0
....zsh              75102           503
...login             21767           0
....zsh              21768           503
.....vim             29141           503
...login             18769           0
....zsh              18770           503
...login             68333           0
....zsh              68334           503
.....vim             29435           503
...login             11721           0
....zsh              11722           503
.....sudo            98701           0
......surge-collect  98702           0
........_surge-collect20 98703           0
...login             11080           0
....zsh              11081           503
.....nstask_updated_s 98698           503
......sleep          98700           503
```

Figure 4.10. Volatility output showing process tree

- **NSAppleScript**

Another way of getting system info on macOS is by using AppleScript. It is a scripting language developed by Apple and allows the user to directly control and automate scriptable macOS applications by interfacing with application event messages [6]. While our example is much simpler, AppleScript makes it possible to interact with the file system, sent text messages through iMessage, launch scripts including with the sh shell, gather volume info, and so on.

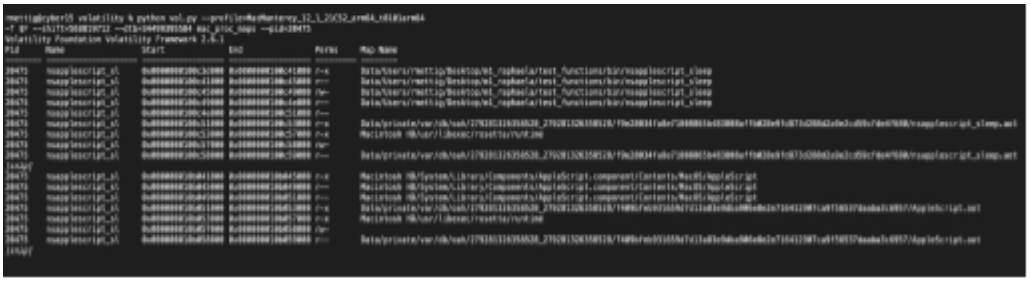


Figure4.11.Volatility-mac proc-mapsoutputshowingreferencestoAOTfilesandAp-pleScriptlibraries

4.1.3. Memory-Only Execution

Memory-only is a particularly interesting and sophisticated attack vector as it does not leave any trace of the malware in the file system. This means that the only way a memory only executable or artifact can be detected or recovered is through memory forensics. The idea behind a malware using memory-only execution is to reduce the chances of it being detected by loading executables, libraries, or bundles from memory as opposed to directly from disk [3].

- **NSCreateObjectFileImageFromMemory**

NSCreateObjectFileImageFromMemory is one of the most common ways to load code from memory on a macOS [3]. In this example, the executable opens an existing file on disk, gets its file size and maps it into memory, then NSCreateObjectFileImageFromMemory loads it into that mapped address. The one restriction encountered was that the program being loaded by NSCreateObjectFileImageFromMemory has to be some kind of Apple Bundle, e.g. compiled as a .app file.

```
rmettig@cyber15 bin % ./nscreateobjmem
Open file...
Getting file size.
Mapping file in memory...
Done.
Create object in memory...
DEBUG -- loadAddress = 0x108c52000, st_size = 16480
Program is succesffully running in memory only at address 0x108c52000
```

Figure 4.12. Example of NSCreateObjectFileImageFromMemory running in the Terminal with the address

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
 -f $F --shift=568819712 --dtb=34499395584 mac_proc_maps --pid=99841
Volatility Foundation Volatility Framework 2.6.1
Pid      Name                 Start              End                Perms     Map Name
-------- -------------------- ------------------ ------------------ --------- --------
[snip]
99841    nscreateobjmem       0x0000000108c52000 0x0000000108c57000 r--       Data/Users/rmettig/Desktop/m1_raphaela/test_functions/bin/bundle
[snip]
```

Figure 4.13. Volatility - NSCreateObjectFileImageFromMemory mapping of the bundle.app executable in the same address listed by the program

As you can see by the figures, the mac_proc_maps plugin that shows the memory mappings of a process, in this case our test program using NSCreateObjectFileImageFromMemory, lists the bundle.app executable that is loaded in the program we ran exactly at the same address that is output into the Terminal. Not listed in the figure, mac_proc_maps also shows multiple references to the test program AOT files, as well as the Rosetta 2 libraries like in the previous example for AppleScript.

4.1.4. Clipboard Monitoring/Copying

Clipboard monitoring is another tactic generally associated with spyware and malware attempting to steal information. An example for this is when dealing with cryptocurrency wallet addresses. Since it is common practice by users to copy and paste wallet addresses, this API can be used to monitor the user's pasteboard for any wallet addresses that show up and either modify its contents or collect that information [2].

- **NSPasteboard**

NSPasteboard is a server that is shared by all running apps on macOS. Part of AppKit, the NSPasteboard objects are the only way applications can communicate with the pasteboard server, and it is used for actions such as copy/paste and communication between apps. The sample application we wrote for NSPasteboard writes a string Hello World to the clipboard from the program and then reveals the string to the terminal output.

```
rmettig@cyber15 bin % ./nspasteboard_sleep
2022-03-22 12:56:11.072 nspasteboard_sleep[86773:11261794] Writing to clipboard...
2022-03-22 12:56:11.076 nspasteboard_sleep[86773:11261794] Done.
2022-03-22 12:56:11.082 nspasteboard_sleep[86773:11261794] Read from clipboard: Hello World
```

Figure 4.14. Example of NSPasteboard running in the Terminal.

In the figures for this example we can see process being dumped by the early fix of the mac_procdump plugin and having that be examined for strings, which give the analyst a decent idea of what the executable is doing.

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
-f $F --shift=568819712 --dtb=34499395584 mac_procdump --dump-dir=out --pid=20500
Volatility Foundation Volatility Framework 2.6.1
Task                        Pid    Address            Path
--------------------------- ------ ------------------ ----
nspasteboard_sle            20500 0x0000000104cc0000 out/task.20500.0x104cc0000.intel.dmp

rmettig@cyber15 volatility % strings out/task.20500.0x104cc0000.intel.dmp
init
generalPasteboard
arrayWithObject:
declareTypes:owner:
setString:forType:
stringForType:
writeToPasteBoard:
readFromPasteBoard
.cxx_destruct
pasteBoard
ClipBoard
@16@0:8
c24@0:8@16
v16@0:8
@"NSPasteboard"
Writing to clipboard...
Hello World
Done.
Read from clipboard: %@
 UnU
```

Figure 4.15. NSPasteboard process dumped and examined for strings in the Terminal.

4.2. Memory Analysis of Rosetta 2

After Rosetta 2 is first installed, the prompt asking for the user's permission will not show up again the next time we try to run an Intel binary. This is important to note because if Rosetta is already installed in the system then an Intel program can be launched in the background without the user's knowledge, especially if the required permissions are already in place. We modified the NSTask example used in section 4.1.2 to launch the addGlobalEventMonitor keylogger from section 4.1.1 to test this, and not

only did it work but the keylogger subprocess persisted after the calling program finished executing and had to be killed separately.

As shown in the previous section, these Intel binaries had little-to-no difficulty in being executed and performing functions associated with malicious behavior. Granted that APIs and functionalities are not only neither inherently good or bad, but they provide necessary tools needed for applications to work properly in a system. It is important to be able to

```
rmettig@cyber15 bin % ./nstask_keylogger
2022-03-23 14:05:33.706 nstask_keylogger[98886:11662913] Program started, launching keylogger
2022-03-23 14:05:33.706 nstask_keylogger[98886:11662913] waiting for new task to exit
2022-03-23 14:05:33.784 globaleventhandler_test[98887:11662917] Accessibility Enabled
2022-03-23 14:05:33.784 globaleventhandler_test[98887:11662917] registering keydown mask
h2022-03-23 14:05:46.688 globaleventhandler_test[98887:11662917] keydown: h
2022-03-23 14:05:47.534 globaleventhandler_test[98887:11662917] keydown: i
i2022-03-23 14:05:53.249 globaleventhandler_test[98887:11662917] keydown: w
w2022-03-23 14:05:53.856 globaleventhandler_test[98887:11662917] keydown: o
o2022-03-23 14:05:55.027 globaleventhandler_test[98887:11662917] keydown: r
r2022-03-23 14:05:55.240 globaleventhandler_test[98887:11662917] keydown: l
l2022-03-23 14:05:55.376 globaleventhandler_test[98887:11662917] keydown: d
```

Figure 4.16. Terminal - NSTask being used to launch keylogger

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
-f $F --shift=568819712 --dtb=34499395584 mac_pstree
Volatility Foundation Volatility Framework 2.6.1
Name                 Pid             Uid
..Terminal           11079           503
...login             75101           0
....zsh              75102           503
...login             21767           0
....zsh              21768           503
.....vim             29141           503
...login             18769           0
....zsh              18770           503
...login             68333           0
....zsh              68334           503
.....vim             29435           503
...login             11721           0
....zsh              11722           503
.....sudo            98891           0
......surge-collect  98892           0
........_surge-collect61 98893          0
...login             11080           0
....zsh              11081           503
.....nstask_keylogger 98886           503
......globaleventhandl 98887          503
```

Figure4.17.Volatility-mac -pstreeoutputshowingkeyloggersubprocessbeingspawned

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
-f $F --shift=568819712 --dtb=34499395584 mac_pstree
Volatility Foundation Volatility Framework 2.6.1
kernel_task          0               0
.launchd             1               0
..mdworker_shared    98919           503
..mdworker_shared    98908           503
..globaleventhandl   98887           503
..globaleventhandl   98865           503
```

Figure 4.18. Volatility - mac pstree output showing keylogger subprocess running on its own detect them and leave it to an experienced analyst to determine whether the behavior is okay and should be allowed, whether it is malicious and needs to be blocked, or whether it is unclear and it needs to be flagged. In the event that the case is the latter, manual analysis of those binaries may be necessary which warrants tools such as Volatility to be able to properly carve out the necessary data from memory and reinforces the need for support of the newer architectures such as M1's ARM64.

For example, here we can see what the /var/db/oah directory containing the AOT files looks like after SIP has been disabled. All of the AOT files for the Intel-based binaries that were ran are listed in there.

```
rmettig@cyber15 279281326358528_279281326358528 % ls -lt | head -20
total 0
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:10 2ace44e09c0aa37f8f479459d66a5d8358a893b492f9e8bf3c10c45b9ed27757
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:08 922ae70144252f391d909a012533bca8bf6fa2dfe00cd5621979e6cc778e1bed
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:08 59a894674c9f175c427d181c8949eec8c1558bed2ed38c68d5428768a98c83af
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:06 129adfd197b2bf1ca64ca32188d17ee79e5be6ab29702966dc145dda88ea65cf
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:05 f9e28034fa8e71808865b483808affb028e9fd873d288d2a9e2cd59cfde4f680
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 16:18 1177992b409596ef82dcdf87c02f0d5c37a1403c6b0f2cba251626b622952f2a
[snip]
rmettig@cyber15 279281326358528_279281326358528 % ls 2ace44e09c0aa37f8f479459d66a5d8358a893b492f9e8bf3c10c45b9ed27757
keylogger.aot
rmettig@cyber15 279281326358528_279281326358528 % ls 922ae70144252f391d909a012533bca8bf6fa2dfe00cd5621979e6cc778e1bed
sleep.aot
rmettig@cyber15 279281326358528_279281326358528 % ls 59a894674c9f175c427d181c8949eec8c1558bed2ed38c68d5428768a98c83af
nstask_sleep.aot
rmettig@cyber15 279281326358528_279281326358528 % ls 129adfd197b2bf1ca64ca32188d17ee79e5be6ab29702966dc145dda88ea65cf
nspasteboard_sleep.aot
rmettig@cyber15 279281326358528_279281326358528 % ls f9e28034fa8e71808865b483808affb028e9fd873d288d2a9e2cd59cfde4f680
nsapplescript_sleep.aot
rmettig@cyber15 279281326358528_279281326358528 % ls 1177992b409596ef82dcdf87c02f0d5c37a1403c6b0f2cba251626b622952f2a
globaleventhandler_test.aot
```

Figure 4.19. /var/db/oah directory listing all the AOT files.

If you notice in the figure of the oah directory, even the /bin/sleep subprocess from section 4.1.2 got its own AOT file, meaning the Intel version was run. This is interesting because it serves as an indicator that the program ran. We don't know whether there are native M1 versions of these binaries from the /bin/ available or whether an Intel process would be able to call an ARM64 subprocess. Further investigation would be required to determine that.

Generally speaking, SIP will be enabled by default, making it impossible to access this directory without a reboot. At that point you would lose any and all information that was contained in RAM if the device is rebooted and would probably have to rely exclusively on disk or potentially network forensics to identify indicators of compromise. However, we have already established in section 4.1.3. why this could be a problem. In the case of the programs that were run to obtain the test memory samples, those binaries were innocuous and did not perform any malicious activity, but it is not uncommon for malware to try to erase its presence on the system or to try to defeat live analysis and reverse engineering methods.

The good news is that, if the Intel binary and its Rosetta process, a.k.a. the AOT file, are still loaded in memory, they can both be extracted for analysis even when SIP is enabled. Memory forensics is the only way to recover the AOT files for an Intel-based memory-only

process when SIP is enabled. For this reason, we have modified the mac_procdump plugin for M1 to dump both the Intel process as well as its respective AOT file from memory.

```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F --shift=568819712
--dtb=34499395584 mac_procdump -p 20584 -D newout
Volatility Foundation Volatility Framework 2.6.1
Task                 Pid    Address           Path
-------------------- ------ ----------------- ----
keylogger             20584 0x0000000102c49000 newout/task.20584.0x102c49000.intel.dmp
keylogger             20584 0x0000000102c4e000 newout/task.20584.0x102c4e000.arm.dmp
```

Figure4.20.Modificationtomac -procdumpplugintodumptheoriginalIntelbinaryand its AOT file.

Lastly, we have also found that even if the original binary is deleted, the AOT file will persist in the oah directory and can be used as an indicator that the Intel binary ran in the system.

Chapter 5. Conclusion and Future Work

5.1. Conclusion

The evolution of malware threats is increasingly challenging traditional analysis methods, highlighting the necessity for adaptable forensic tools capable of handling diverse computing environments. Apple's M1 chip, based on ARM64 architecture, introduced a need for specialized research to enhance support within tools like the Volatility Framework. This research sought to address these challenges through various approaches. First, it examined the procedural adjustments necessary for conducting memory forensics on M1 devices, including identifying supported tools and configuring the environment for memory sample acquisition and Volatility execution. Second, it explored the Objective-C API for Intel binaries on ARM—often exploited by malware—resulting in curated memory samples to validate Volatility plugin development and testing. Third, an analysis of the Rosetta 2 environment revealed potential forensic artifacts specific to M1 memory analysis. This effort also included multiple updates to Volatility 2 plugins, supporting the M1 architecture and investigating whether existing macOS malware could run on M1 systems via Rosetta.

5.2. Future Work

Significant work remains to be done to fully adapt forensic tools for Apple's M1 architecture. Although some Volatility plugins, such as mac_rosetta, and updates to plugins like mac_procdump, now allow for Intel binary and AOT file dumping on M1, many macOS plugins still require adaptation for compatibility with this new architecture. Additionally, the Rosetta 2 runtime offers a largely unexplored avenue for potential attack vectors and forensic artifact discovery. Continued investigation in this area may yield further insights and broaden the forensic toolkit for M1 devices.

References

1. Parallels Knowledge Base, "Install macOS Monterey 12 virtual machine on a Mac with Apple M1 chips," 2021. Available: Parallels Knowledge Base
2. Lawrence Abrams, "Clipboard hijacker malware monitors 2.3 million bitcoin addresses," 2018. Available: Bleeping Computer
3. Stephanie Archibald, "Running executables on macOS from memory," 2017. Available: Blackberry Blog
4. @08Tc3wBB, "Kernel exploitation on Apple's M1 chip," 2021. Available: Objective by the Sea
5. Apple, "Apple to use Intel microprocessors beginning in 2006," 2006. Available: Apple Newsroom
6. Apple, "Applescript Language Guide," 2016. Available: Apple Developer
7. Apple, "About System Integrity Protection on your Mac," 2019. Available: Apple Support
8. Apple, "Apple unleashes M1," 2020. Available: Apple Newsroom

9. Apple, "Introducing M1 Pro and M1 Max: The most powerful chips Apple has ever built," 2021. Available: Apple Newsroom