IMPROVING MEMORY FORENSICS CAPABILITIES ON APPLE M1 COMPUTERS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Computer Science and Engineering

by
Raphaela Santos Mettig Rocha
B.S., Louisiana State University, 2020
May 2022

Acknowledgments

I could not have accomplished this work without the support from those around me.

First I'd like to thank my advisor Dr. Golden Richard for your continued mentorship, insight, and expertise throughout the years. You've helped me navigate both the academic and industry worlds from early on and I will be forever in your debt.

I'd like to particularly acknowledge Andrew Case for your invaluable guidance during the research process and for being such an inspiring professional for myself and others in the lab. I'd like to thank Dr. Nash Mahmoud and Dr. Gerald Baumgartner for being on my committee and for also being trusted mentors.

To my parents, who have always been my greatest supporters, to my friends, who became my family away from home, and to my therapist, for helping me get through this part of my life in one piece. It's because of you all that I've made it this far.

I would also like to acknowledge my colleagues at the LSU Applied Cybersecurity Lab for keeping me company and helping me stay sane during the long hours of work. You all constantly push me to be a better person and researcher.

And finally, to Oliver, my cat. I do this so you can have a better life.

Table of Contents

List of Tables

List of Figures

Abstract

Malware threats are rapidly evolving to use more sophisticated attacks. By abusing rich application APIs such as Objective-C's, they are able to gather information about user activity, launch background processes without the user's knowledge as well as perform other malicious activities. In some cases, memory forensics is the only way to recover artifacts related to this malicious activity, as is the case with memory-only execution. The introduction of the Rosetta 2 on the Apple M1 introduces a completely new attack surface by allowing binaries of both Intel x86_64 and ARM64 architecture to run in userland. For this reason it is important that forensic analysis tools are able to properly identify indicators of malicious activity in a memory sample as well as include support for new platforms as soon as possible. In this paper, I present a memory analysis of the Rosetta 2 runtime using the Volatility Framework, a set of curated memory samples to help validate Volatility plugins and algorithms, and discuss new contributions to the Volatility support of the M1 platform.

Chapter 1. Introduction

The primary objective of this research is to contribute to the capabilities of memory forensics techniques on the new Apple M1 ARM-based chip.

1.1. Memory Forensics

Memory forensics is a set of tools and techniques used to perform analysis on volatile memory (RAM) usually during an incident response investigation or for research purposes. It is a powerful tool for any digital forensic analyst because it provides unparalleled insight into a computer's state at the time the memory capture is taken. With memory forensics it is possible to recover artifacts such as running processes and recently-terminated processes, active network connections, open files, registry information, and much more. In some cases, such as when dealing with memory-only malware or encrypted code and data, memory analysis is the only way to perform a complete investigation of that system.

Once a memory sample is acquired, a tool capable of parsing that information and extracting the data is necessary in order to perform the analysis. The Volatility Framework for memory forensics is the most widely used tool by cybersecurity professionals to analyze memory samples. With malware threats becoming more sophisticated every day, it is imperative that an analysis framework like Volatility is capable of supporting new platforms as soon as possible. This research is focused on the Volatility Framework and aims to expand Volatility's support of the new Apple M1 chip by producing memory samples that can be tested against to update the existing plugins and introduce new plugins.

A real-life example of where memory forensics would be necessary in an investigation would be in the case of Duqu [19]. The threat actor group Duqu is known for being behind one of the most sophisticated malware attacks back in 2011 and 2015. In 2015 Duqu 2.0 was found by Kaspersky Labs during an internal sweep. It is a worm very similar to Stuxnet's, which took down Iranian nuclear plants in 2007, but with a more selective means of propagation. One of the reasons it was very difficult to detect is that both of its payloads, a backdoor and a fully-featured Command & Control platform, were entirely in-memory. Another example is when dealing with attacks that leverage Cobalt Strike, a security tool used for emulating adversary attacks on the network. Cobalt Strike often gets abused by real attackers, and all of its payloads are launched memory-only from the operator's system to the victim's [23].

1.2. The Apple M1

In November 2020 Apple announced their transition to a processor based on the ARM64 architecture and introduced the M1, also commonly known as Apple Silicon. The M1 is considered a system on a chip, as it integrates several components into a single chip, including a unified memory architecture which, among other components, greatly helps increase its performance [8]. According to Apple's press release about the M1 in November 2020, the original M1 had the world's best CPU performance per Watt, the world's fastest integrated graphics, a dedicated machine learning engine, and much more. In October 2021, Apple introduced the M1 Pro and M1 Max that are even more powerful with features up to a 10-core CPU, 32-core GPU, and 64GB of unified memory [9]. Now, in March 2022, they have announced the M1 Ultra, which can be configured with up to 128GB of high-bandwidth, low-latency unified memory that can be accessed by the 20-core CPU, 64-core GPU, and 32-core Neural Engine [11]. Needless to say, the M1 has a lot of potential in terms of technology and appears to be on the way to becoming a commercial success.

However, this transition from an Intel x86-64 architecture to ARM64 introduced several changes to the software landscape within the Apple ecosystem, the biggest one being that most of the software that was previously available on Intel-based computers not being compatible with the new ARM architecture. To address this compatibility issue, Apple also introduced the Rosetta 2 runtime. Rosetta 1 came out in 2006 when Apple first switched from PowerPC to Intel architecture, and now Rosetta 2 is available in this new transition[5]. Rosetta is software that acts as a translation layer between two architectures. It is included in the operating system when they are working on transitioning to a new chip, and it eventually gets phased out. The idea is that once an x86 binary is run on the M1, the Rosetta 2 daemon launches and performs the translation at runtime. While there is still not much public information about its inner workings, I will talk about some of its technical details in Chapter 3.

1.3. Research Importance

There are a few reasons why investigation of Rosetta 2 from a memory forensics perspective is relevant. The first is that the Rosetta 2 layer's ability to run x86 binaries potentially introduces a new attack surface for malware on top of expanding the previous one. This occurs as all previous malware samples and techniques are potentially reusable on M1 without modification from their Intel-compiled form. At this time there are not much deep technical research published given how new both M1 and Rosetta 2's technology are, so it is important to try to better understand what the risks associated with introducing this surface is. I will go more in depth about this issue in the background section. Second – historically speaking, memory forensics research has mostly focused on detecting and analyzing kernel-level malware, leaving the userland side relatively unattended. With vendors increasingly restricting access to kernel-level functionalities, targeting userland activities becomes more attractive for malware developers. This alone warrants a dedicated research effort to expand the support for analysis of new platforms [16]. This becomes clear in Chapter 3 where I go over the necessary preparations on the system to test the common API functions used in Objective-C. The final reason is that memory forensics is used on a daily basis to respond to real-world incidents that greatly affect the privacy and security of users and organizations as well as in a variety of legal settings. For this reason, it is imperative that memory forensics tools be tested and validated as thoroughly as possible for all situations for which the tools may be used. During this research process, programs were developed that attempted to abuse APIs available on Intel systems for malicious tasks, such as keylogging and clipboard manipulation, to see if the same APIs were available and functional through Rosetta 2. Where available and functional, these APIs immediately become usable for malware targeting M1s without extra research by the malware authors. This threat also highlights the important of our M1-specific research.

To achieve the desired goals of this research project, the first step was to assess what existing tools are currently able to detect and analyze malware on this platform, as well as to determine where to aim future research efforts. By using our developed suite of proof-of-concept malware, memory forensics researchers can test precise techniques against M1 devices.

1.4. Outline

This paper is organized as follows: Chapter 2 covers the Background, which will go more in depth on the Volatility Framework for memory forensics, the current state of memory forensics on the M1, discuss previous work that explains the focus of this paper on userland malware, and give more detail on how the Rosetta 2 translation layer works. In Chapter 3 I will outline in detail how the environment was setup for this work and how memory samples were acquired from the M1 system. Chapter 4 covers the API tests that were done and the analysis of Rosetta and discuss them. Chapter 5 covers other contributions done in parallel to this work. Lastly, chapter 6 concludes this work and will and talk about future works.

Chapter 2. Background and Related Works

This chapter will go into detail on the related works that were relevant to this research, as well as provide an overview for the concepts that are necessary to understand this work.

2.1. Volatility Framework Overview

The Volatility Framework for memory forensics is a free and open-source collection of tools that is widely used by professionals in the cybersecurity industry. It is implemented in Python 2.7, and its primary focus is on forensics, incident response, and malware analysis [21]. The book Art of Memory Forensics by Ligh et al. [21] is the primary reference guide for Volatility and memory forensics. Published in 2014, it provides the necessary background information on system internals concepts necessary for doing memory analysis as well as explains how most of the Volatility plugins work. The book contains individual sections dedicated to Windows, Linux, and Mac which go over the specific details for how each set of plugins work with their respective operating systems.

One relevant feature about Volatility is the ability for any user to expand on the framework through the development of plugins. Volatility parses the memory sample, and each plugin is a different program that extracts specific information from a memory dump, such as process lists and trees, active network connections and ports, open files and their respective processes, and much more. Plugins are also able to scan the memory sample for known data structures, strings, encrypted data, YARA signatures, as well as dump processes and files carved from memory. However in order to develop a plugin with these capabilities, it is necessary to have a set of trusted memory samples that are known to have the artifacts the plugin will be looking for in order to validate whether they're working properly or not. One of the main goals of this work is to generate a set of samples containing M1-related artifacts that will be used to test both new and modified plugins for the platform.

2.2. Memory Forensics on Intel-based macOS

Historically, memory forensics research has focused on detecting and analyzing kernel-level malware [16]. The kernel is an interesting target for malware as it provides the high privilege to control system functionality. For this reason, operating system developers and security researchers are constantly looking to increase kernel security. This focus on kernel security led to userland attacks becoming much more attractive to malware authors as the security research and development on that space have not been met with equal effort, leading to a gap in analysis tools capable of detecting malware functioning in that space. Case et al. [16] published a paper in 2016 describing a new suite of Volatility plugins capable of detecting Objective-C methods that are commonly abused by malware in macOS memory. Later in 2019, Manna et al. [22] expanded on Case's work and did a more in-depth analysis of the Objective-C runtime and its related data structures. Both these papers served as a foundation for selecting the Objective-C API class functions that will be used to generate the memory samples for validation testing. I will talk more about them in Chapter 4.

2.3. M1 Analysis

Prior to the start of our research efforts for the M1, there still had not been many deep technical analyses performed on the new architecture, given how new it is to the market. Below is a discussion of work most closely related to this research.

2.4. Memory Forensics on M1-based macOS

Previous work by Joshua Duke done in 2021 [17] did an extensive analysis on all the standard macOS Volatility plugins. Duke's work ran the plugins on the M1 and documented any differences in their output or errors that occurred because of the new architecture. With this information, the Volatility developers are able to make the necessary fixes to the existing plugins.

2.4.1. Other M1 Research Efforts

Other notable related works include a presentation by researchers from Corellium at BlackHat 2021 about their efforts to reverse engineer the M1 at the hardware level [28], and a presentation by researchers from ZecOps at the Objective By the Sea conference on M1 exploitation methods [4].

2.5. Rosetta 2

As previously mentioned, when Apple introduced the M1 they also released a new version of their Rosetta translation layer, Rosetta 2. The goal for Rosetta 2 is to ease the transition to Apple silicon by allowing software that ran on the previous architecture to run on the M1 while their developers worked on a universal binary or native version of the app [10].

There are currently two ways to run Intel binaries on the M1: through Universal 2 binary, or through Rosetta 2. The Universal 2 binary is a Mach-O file, the standard executable format for Apple operating systems, and it contains code for both architectures. The source code is compiled twice, once for each architecture, and then combined into a single binary with the `lipo` command line tool [12]. Then there's Rosetta 2. According to the Apple documentation, Rosetta can translate most Intel-based apps except for any kernel extensions and Virtual Machine apps that virtualize x86_64, such as VMware [10].

As far as research goes, Project Champollion by Koh M. Nakagawa [20] is the closest analysis done to this work. Nakagawa decided to explore Rosetta 2 as a potential attack surface for exploitation and wrote a deep analysis in a series of blog posts entitled "Project Champollion". His efforts outline the internal structures for the Rosetta 2 runtime and his contributions include a patch for the Ghidra decompiler [24] to be able to correctly parse AOT files, as Apple's implementation of ARM64 has been slightly modified [14].

AOT stands for Ahead-of-Time translation because the x86_64 binary is translated into a special type Mach-O file with the extension `.aot` before running [15, 20]. Nakagawa discovered that the daemon that handles the translation process is called `oahd` and is invoked when the Intel binary tries to execute. `oahd` will check in the `/var/db/oah` directory whether there already exists an AOT file for that specific binary. This directory is protected by Apple's System Integrity Protection (SIP) [7] and cannot be accessed directly by any user until SIP is turned off. If a binary is running for the first time this translation will be performed prior to its execution by the `oahd-helper`, which will translate Intel code into ARM code and write it to the AOT file [20].

Chapter 3. Memory Acquisition and Environment Setup

Now we will go over the specific tools that were used, as well as explain how the environment was set up for this research.

3.1. Memory Acquisition

This was one of the more difficult parts of the setup stage. The main component necessary to perform memory analysis is getting a memory sample from the target system. On Intel-based systems there are two main methods an investigator may use to acquire the memory for analysis. The first is through virtualization software (a.k.a. virtual machines or VMs) such as VMWare or Parallels, which allow for snapshots of the guest operating system to be taken and those include a file with the memory contents. The second is using a software-based acquisition tool that is installed on the host and is able to dump the physical memory that is visible by the operating system.

Unfortunately at this time neither VMware nor Parallels are able to fulfill our needs for analysis. Memory analysis requires VMs to be able to be suspended and take system snapshots, and neither feature is currently available for the M1. VMware Fusion is only available in a beta state and might not have a stable version available until later in Spring 2022, and Parallels, which has a stable release for M1, does not allow suspending and snapshotting guests [18, 1, 26]. It appears this will change in the future, but this means that for the time being we need a different option.

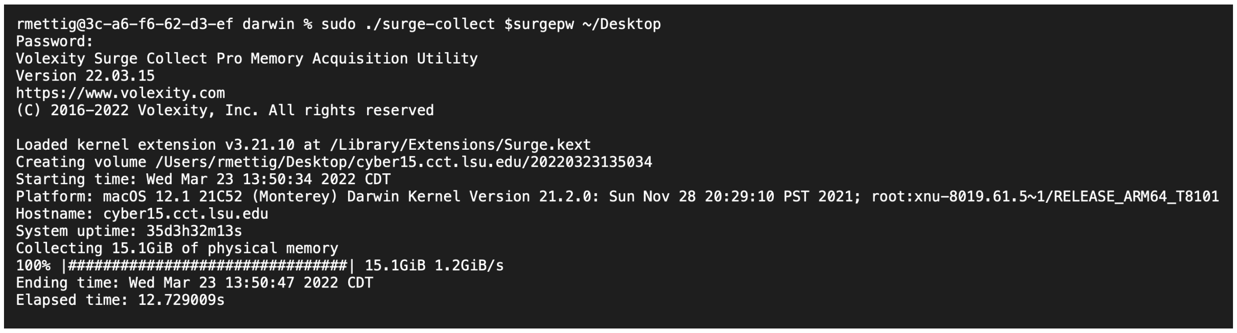
The alternative was to resort to a software-based acquisition tool. Our group was provided with a temporary license to use Surge Collect Pro [29], which can successfully generate the memory samples needed from the M1 system. I would like to thank Volexity for kindly providing a temporary license to the full software. This research would not have been possible without it as all of the memory samples used in this work are taken directly from the host machine. No VMs were used in this part of the research.

3.2. Environment Setup

The main system used for analysis was a 2021 Apple 24-inch iMac, running macOS 12.1 Monterey on the M1 chip, with 16GB of memory and 1TB of storage. The code development portion was done and also tested on a 2019 Apple MacBook Pro, running macOS 12.1 Monterey on a 8-core Intel Core i9.

Development work was required for this project. All of the Objective-C binaries that will be described in the next chapter were built and tested with XCode v13.2.1 on the 2019 Intel Macbook Pro.

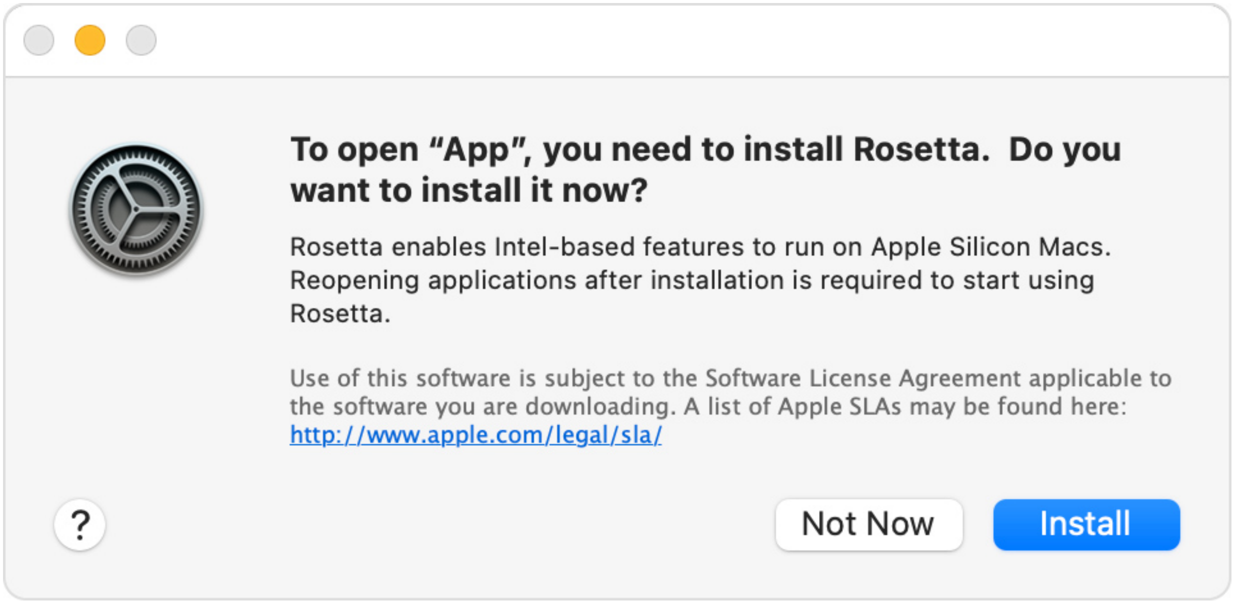
In order to run Surge Collect Pro, a custom kernel extension (kext) had to be installed on the machine. Kexts are bundles that have the elevated privileges to perform low-level tasks in the kernelspace [13]. Surge would not have access to the system's physical memory otherwise, and the extension is provided with it. However, as of macOS 11 and later, loading a kernel extension requires the user to boot into Recovery Mode, set the security level to Reduced security, and allow the loading of third-party kexts. After rebooting, the user also has to give permissions under the Security and Privacy System Settings to allow the kext to be loaded by the system, and then reboot again [13]. This process can also be performed at scale in enterprises through the use of MDM software. Once the Surge kext has been loaded, its command line tool provides a very easy method for acquiring memory from the system.



```
rmettig@3c-a6-f6-62-d3-ef darwin % sudo ./surge-collect $surgepw ~/Desktop
Password:
Volexity Surge Collect Pro Memory Acquisition Utility
Version 22.03.15
https://www.volexity.com
(C) 2016-2022 Volexity, Inc. All rights reserved

Loaded kernel extension v3.21.10 at /Library/Extensions/Surge.kext
Creating volume /Users/rmettig/Desktop/cyber15.cct.lsu.edu/20220323135034
Starting time: Wed Mar 23 13:50:34 2022 CDT
Platform: macOS 12.1 21C52 (Monterey) Darwin Kernel Version 21.2.0: Sun Nov 28 20:29:10 PST 2021; root:xnu-8019.61.5~1/RELEASE_ARM64_T8101
Hostname: cyber15.cct.lsu.edu
System uptime: 35d3h32m13s
Collecting 15.1GiB of physical memory
100% |################################| 15.1GiB 1.2GiB/s
Ending time: Wed Mar 23 13:50:47 2022 CDT
Elapsed time: 12.729009s
```

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

Next, we need to install Rosetta 2 as it does not come pre-installed with macOS. Once a user attempts to run an Intel binary for the first time, they will be prompted to install it.


To open "App", you need to install Rosetta. Do you want to install it now?

Rosetta enables Intel-based features to run on Apple Silicon Macs. Reopening applications after installation is required to start using Rosetta.

Use of this software is subject to the Software License Agreement applicable to the software you are downloading. A list of Apple SLAs may be found here: http://www.apple.com/legal/sla/

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

Lastly, we also needed to disable SIP as it protects the oah folder that contains the AOT files. SIP restricts access to certain system directories even by the root user, only allowing processes signed by Apple to modify and write to these system files [7]. To do that, I had to restart the computer in Recovery Mode once again and enter `csrutil disable` in the Terminal, then reboot it again.

3.2.1. Volatility for M1

For this project, we used our currently internal version of the Volatility framework v2.6.1, which contains many fixes and new additions so that the framework can support the ARM64 architecture and M1 analysis.

New profiles for M1 kernels were necessary to get Volatility working. Volatility works by being able to reconstruct the state of the kernel at the time of acquisition, and profiles contain all the symbols and type information for a given kernel version. Volexity also kindly provided the necessary profiles necessary for this project.

Another requirement to get Volatility working was having a Volatility address space for ARM64. Address spaces support emulating the memory management unit from the CPU and performing necessary address translation from virtual addresses to the physical offsets of the pages in the memory samples. Tsahi Zidenberg [30] added an ARM64 address space to his fork of Volatility, and that is currently being used as the basis for address translation under Volatility on M1.

Lastly, when using Volatility on macOS memory samples, it is necessary to provide the information for the kernel address space layout randomization (ASLR) shift and the physical offset of the kernel page tables (DTB), which change between each reboot of the machine. Thankfully, that information is also provided in a JSON file that Surge creates after it collects memory.

Other additions will be discussed later in the Memory Analysis Chapter and others in the Additional Contributions Chapter.

Chapter 4. Memory Analysis

The memory analysis for this research breaks down into two overlapping components: testing functions from the Objective-C API that are commonly abused by malware, and exploring the Rosetta 2 runtime for new analysis artifacts. The goal is to verify whether these API functions still work as expected and generate a curated set of memory samples that can be used to validate tests for Volatility plugins that need to be fixed or that will be developed.

4.1. Testing the Objective-C API

After getting the environment setup, the first step was to prepare the binaries that would be needed to gather our memory samples. The functions chosen in table 4.1. are all known to have been abused by malware found in the wild and were selected from the lists provided in the macOS malware papers that were discussed in 2.2. [16, 22]. The majority of them are `.m` Objective-C files, with two examples being compiled as `.cpp` C++ files.

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

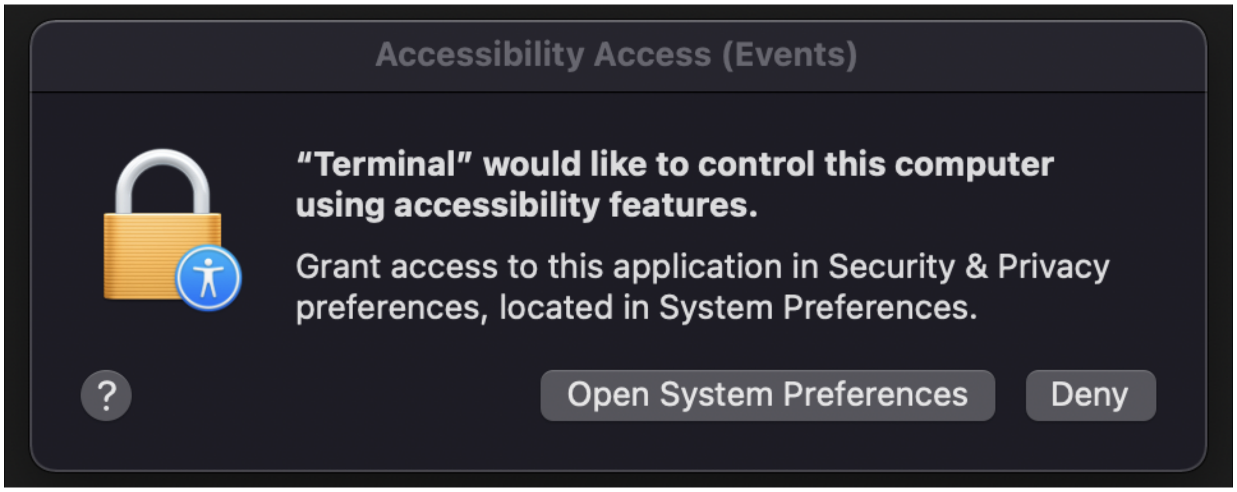
Name	Reason
addGlobalMonitorForEventsMatchingMask	Keylogging
NSEvent	Monitoring devices (keyboard, mouse, etc)
CGEventTapCreate	Keylogging
CGEventTapEnable	Keylogging
NSTask::launch	Allows running executables
NSAppleScript	Run scripts and data gathering
NSCreateObjectFileImageFromMemory	Memory-only execution
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 [22]. 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.


Accessibility Access (Events)

"Terminal" would like to control this computer using accessibility features.

Grant access to this application in Security & Privacy preferences, located in System Preferences.

Figure 4.1. Terminal prompt requesting Acessibility Permissions.

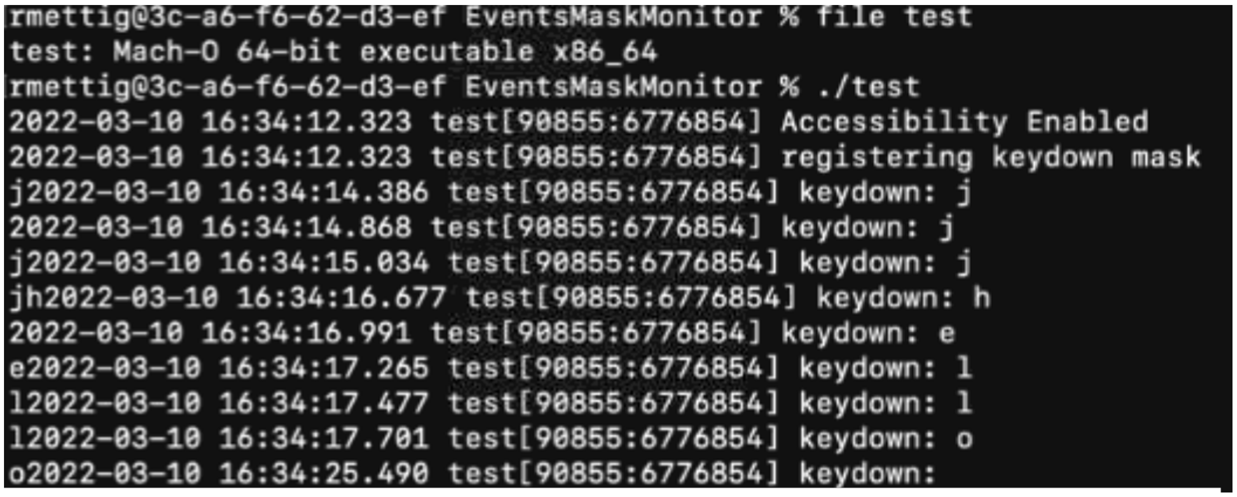
Since the permissions were reset between running each keylogger, both triggered this notification. However, if the Terminal app already has permission prior to running the executable then it is possible to run the program without the user's knowledge.

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.

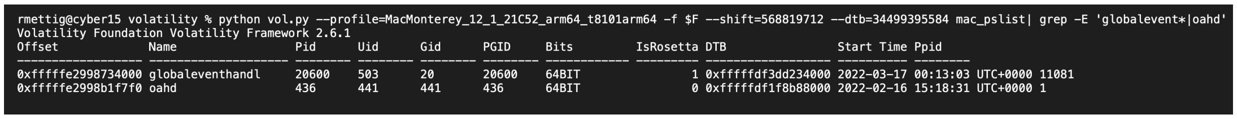


```
rmettig@3c-a6-f6-62-d3-ef EventsMaskMonitor % file test
test: Mach-O 64-bit executable x86_64
rmettig@3c-a6-f6-62-d3-ef EventsMaskMonitor % ./test
2022-03-10 16:34:12.323 test[90855:6776854] Accessibility Enabled
2022-03-10 16:34:12.323 test[90855:6776854] registering keydown mask
j2022-03-10 16:34:14.386 test[90855:6776854] keydown: j
2022-03-10 16:34:14.868 test[90855:6776854] keydown: j
j2022-03-10 16:34:15.034 test[90855:6776854] keydown: j
jh2022-03-10 16:34:16.677 test[90855:6776854] keydown: h
2022-03-10 16:34:16.991 test[90855:6776854] keydown: e
e2022-03-10 16:34:17.265 test[90855:6776854] keydown: l
l2022-03-10 16:34:17.477 test[90855:6776854] keydown: l
l2022-03-10 16:34:17.701 test[90855:6776854] keydown: o
o2022-03-10 16:34:25.490 test[90855:6776854] keydown:
```

Figure 4.2. Volatility - globaleventhandler running on the command line

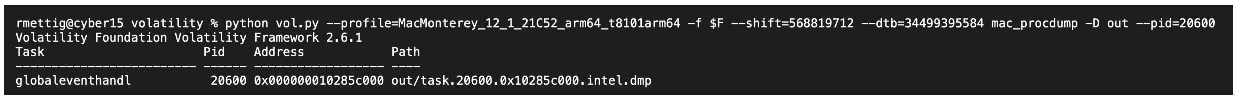
One thing that was added to this example was a check for the Accessibility Permissions. It was interesting to note that, if the permissions for the Terminal were disabled, the program would still run but the event monitor was not able to capture the input. In the end the only memory sample taken was 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.



```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F --shift=568819712 --dtb=34499395584 mac_pslist| grep -E 'globalevent*|oahd'
Volatility Foundation Volatility Framework 2.6.1
Offset             Name                 Pid      Uid      Gid      PGID     Bits         IsRosetta DTB                Start Time Ppid
------------------ -------------------- -------- -------- -------- -------- ------------ --------- ------------------ ---------- --------
0xfffffe2998734000 globaleventhandl     20600    503      20       20600    64BIT                1 0xfffffdf3dd234000 2022-03-17 00:13:03 UTC+0000 11081
0xfffffe2998b1f7f0 oahd                 436      441      441      436      64BIT                0 0xfffffdf1f8b88000 2022-02-16 15:18:31 UTC+0000 1
```

Figure 4.3. Volatility output showing the globalhandler 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
```

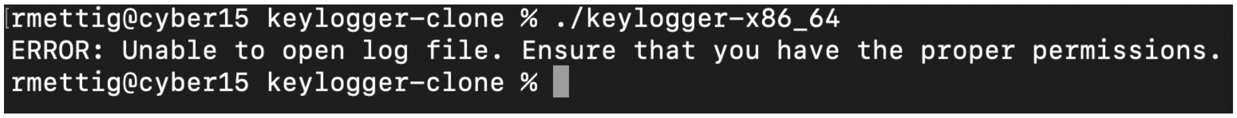
Figure 4.4. Volatility output showing the globaleventhandler binary dumped form memory

- **CGEventTapCreate / CGEventTapEnable**

`CGEventTapCreate` and `CGEventTapEnable` are both methods from Apples CoreGraphics framework. `tapCreate` allows for an event tap to be created, and if sucessful, `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.

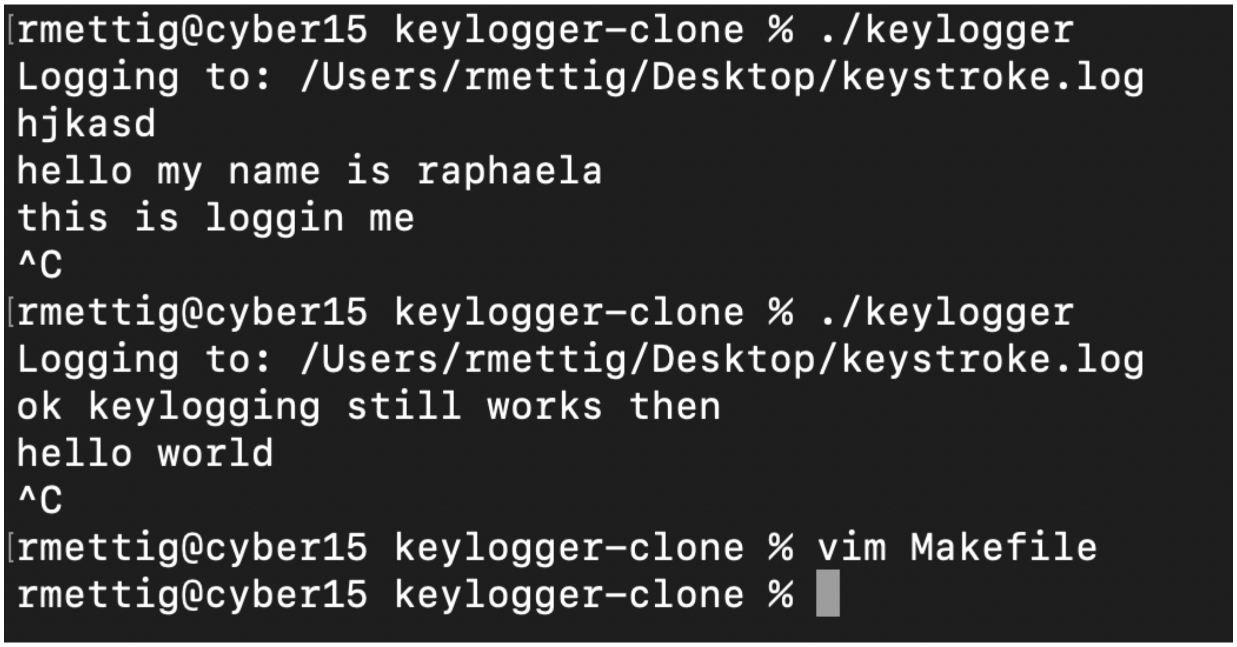
For this example, we used a proof of concept keylogger written by Casey Scarborough [27] that is written in C and that was able to compile and run after a few minor modifications. This specific example will actually output to a file that will be stored on disk. One of the modifications necessary was 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



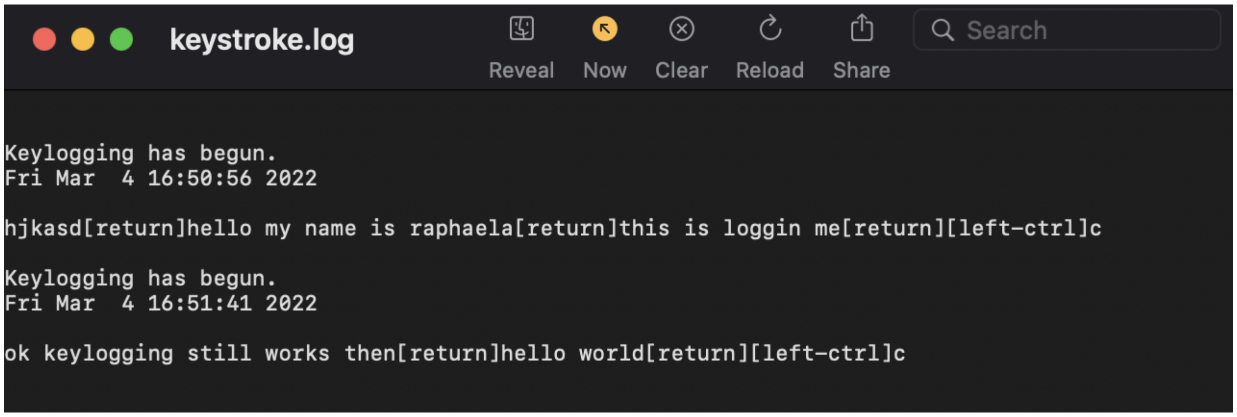
```
rmettig@cyber15 keylogger-clone % ./keylogger-x86_64
ERROR: Unable to open log file. Ensure that you have the proper permissions.
rmettig@cyber15 keylogger-clone %
```

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 %
```

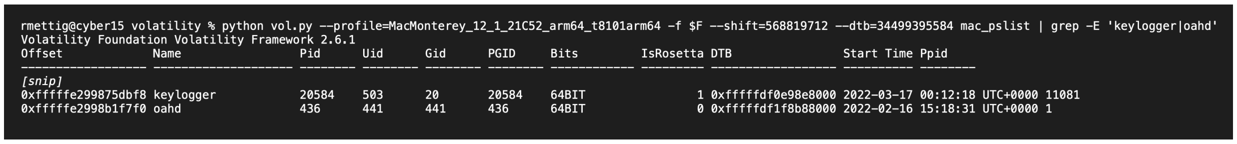
Figure 4.6. CGEventTap Keylogger Running



```
Keylogging has begun.
Fri Mar  4 16:50:56 2022

hjkasd[return]hello my name is raphaela[return]this is loggin me[return][left-ctrl]c

Keylogging has begun.
Fri Mar  4 16:51:41 2022

ok keylogging still works then[return]hello world[return][left-ctrl]c
```

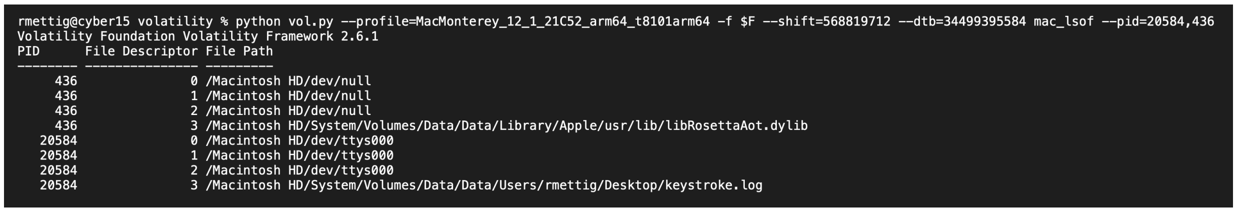
Figure 4.7. CGEventTap Keylogger Logfile

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

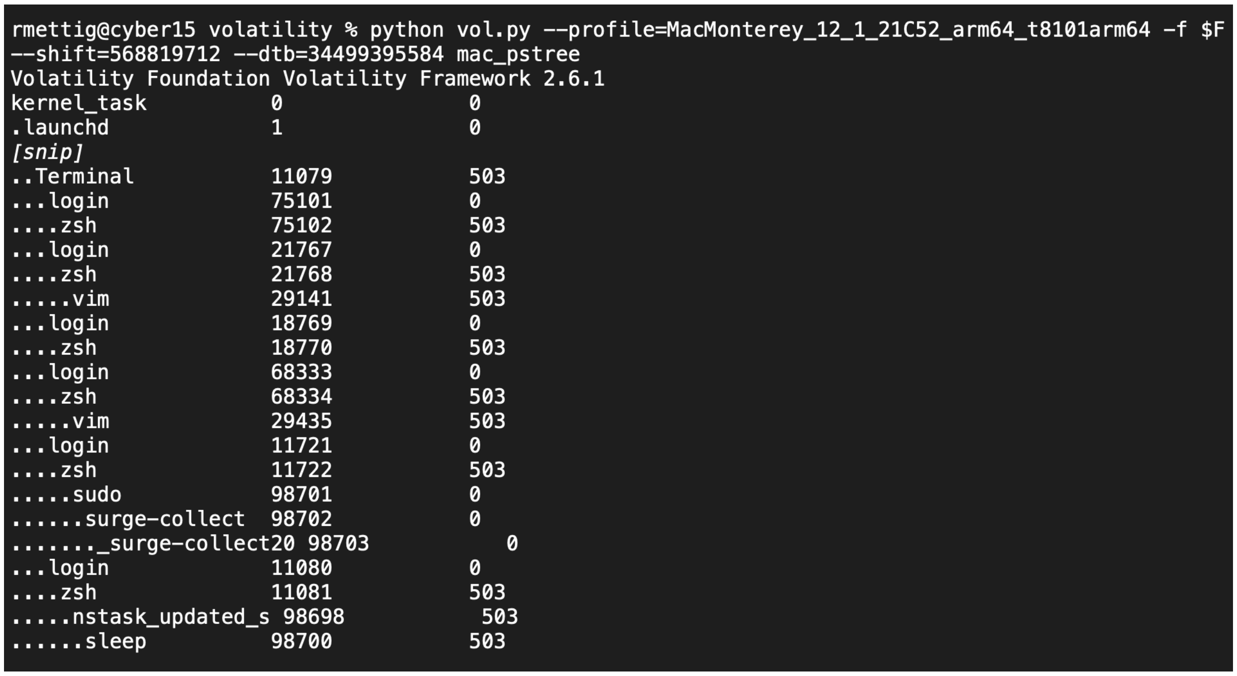
Launching executables is another important vector that we need to pay attention to. It is a common technique for malware to use downloaders as an initial infection vector, which check their running environment for information before doing anything else. An example of this is Proton.B malware for macOS 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 listed as a child process for `nstask_updated_sleep`.

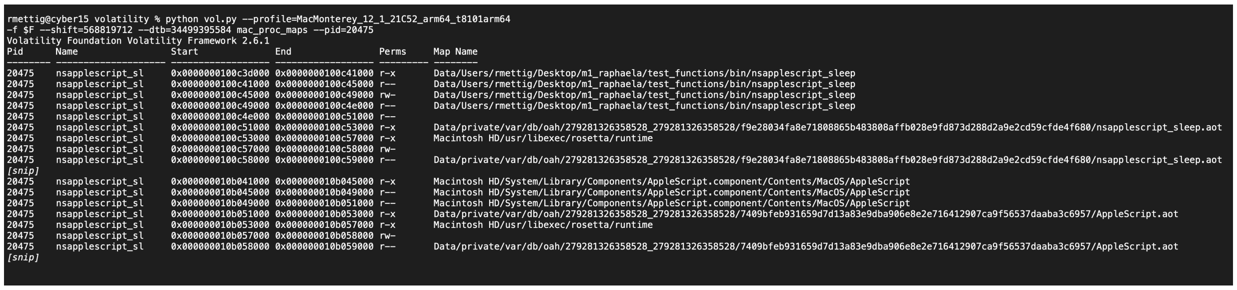


```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64 -f $F
--shift=568819712 --dtb=34499395584 mac_pstree
Volatility Foundation Volatility Framework 2.6.1
kernel_task          0              0
.launchd             1              0
[snip]
..Terminal           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.



```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
-f $F --shift=568819712 --dtb=34499395584 mac_proc_maps --pid=20475
Volatility Foundation Volatility Framework 2.6.1
Pid      Name                 Start              End                Perms     Map Name
-------- -------------------- ------------------ ------------------ --------- --------
20475    nsapplescript_sl     0x0000000100c3d000 0x0000000100c41000 r-x       Data/Users/rmettig/Desktop/m1_raphaela/test_functions/bin/nsapplescript_sleep
20475    nsapplescript_sl     0x0000000100c41000 0x0000000100c45000 r--       Data/Users/rmettig/Desktop/m1_raphaela/test_functions/bin/nsapplescript_sleep
20475    nsapplescript_sl     0x0000000100c45000 0x0000000100c49000 rw-       Data/Users/rmettig/Desktop/m1_raphaela/test_functions/bin/nsapplescript_sleep
20475    nsapplescript_sl     0x0000000100c49000 0x0000000100c4e000 r--       Data/Users/rmettig/Desktop/m1_raphaela/test_functions/bin/nsapplescript_sleep
20475    nsapplescript_sl     0x0000000100c4e000 0x0000000100c51000 r--
20475    nsapplescript_sl     0x0000000100c51000 0x0000000100c53000 r-x       Data/private/var/db/oah/279281326358528_279281326358528/f9e28034fa8e71808865b483808affb028e9fd873d288d2a9e2cd59cfde4f680/nsapplescript_sleep.aot
20475    nsapplescript_sl     0x0000000100c53000 0x0000000100c57000 r-x       Macintosh HD/usr/libexec/rosetta/runtime
20475    nsapplescript_sl     0x0000000100c57000 0x0000000100c58000 rw-
20475    nsapplescript_sl     0x0000000100c58000 0x0000000100c59000 r--       Data/private/var/db/oah/279281326358528_279281326358528/f9e28034fa8e71808865b483808affb028e9fd873d288d2a9e2cd59cfde4f680/nsapplescript_sleep.aot
[snip]
20475    nsapplescript_sl     0x000000010b041000 0x000000010b045000 r-x       Macintosh HD/System/Library/Components/AppleScript.component/Contents/MacOS/AppleScript
20475    nsapplescript_sl     0x000000010b045000 0x000000010b049000 r--       Macintosh HD/System/Library/Components/AppleScript.component/Contents/MacOS/AppleScript
20475    nsapplescript_sl     0x000000010b049000 0x000000010b051000 r--       Macintosh HD/System/Library/Components/AppleScript.component/Contents/MacOS/AppleScript
20475    nsapplescript_sl     0x000000010b051000 0x000000010b053000 r-x       Data/private/var/db/oah/279281326358528_279281326358528/7409bfeb931659d7d13a83e9dba906e8e2e716412907ca9f56537daaba3c6957/AppleScript.aot
20475    nsapplescript_sl     0x000000010b053000 0x000000010b057000 r-x       Macintosh HD/usr/libexec/rosetta/runtime
20475    nsapplescript_sl     0x000000010b057000 0x000000010b058000 rw-
20475    nsapplescript_sl     0x000000010b058000 0x000000010b059000 r--       Data/private/var/db/oah/279281326358528_279281326358528/7409bfeb931659d7d13a83e9dba906e8e2e716412907ca9f56537daaba3c6957/AppleScript.aot
[snip]
```

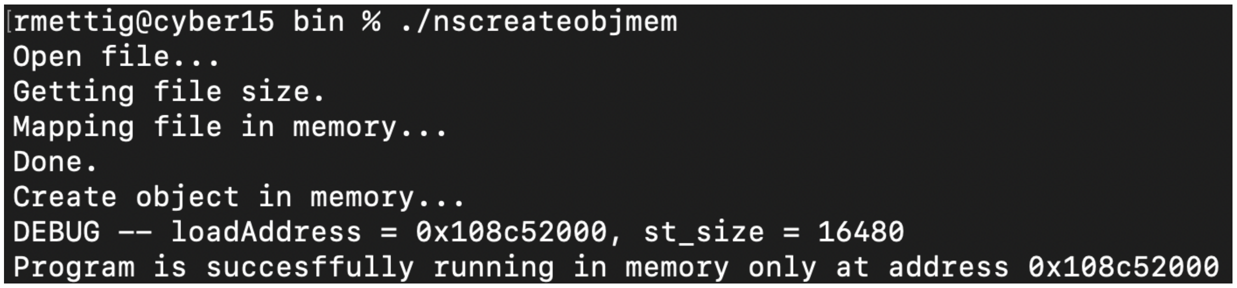
Figure 4.11. Volatility - mac_proc_maps output showing references to AOT files and AppleScript libraries

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].

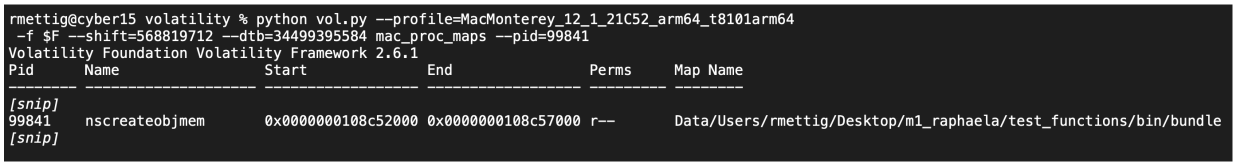
- **NSCreateObjectFileImageFromMemory**

`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

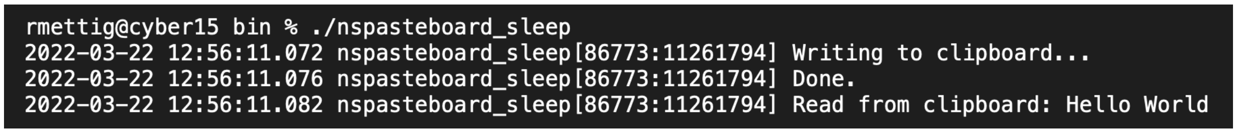
As you can see by the figures, the `mac_proc_maps` plugin that shows the memory mappings of a process, in this case our test program using `NSCreateObjectFileImageFromMemory`, lists the bundle.app executable that is loaded in the program we ran exactly at the same address that is output into the Terminal. 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**

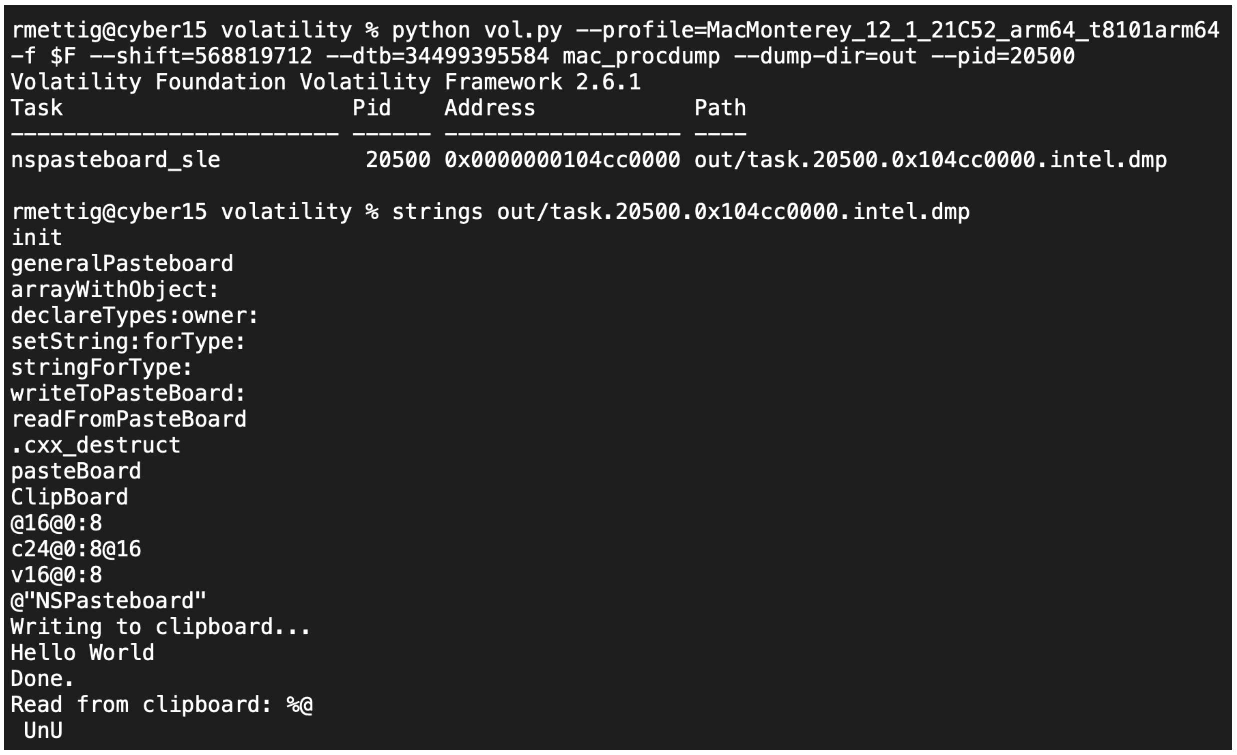
`NSPasteboard` is a server that is shared by all running apps on macOS. Part of AppKit, the `NSPasteboard` objects are the only way applications can communicate with the pasteboard server, and it is used for actions such as copy/paste and communication between apps. The sample application we wrote for 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.



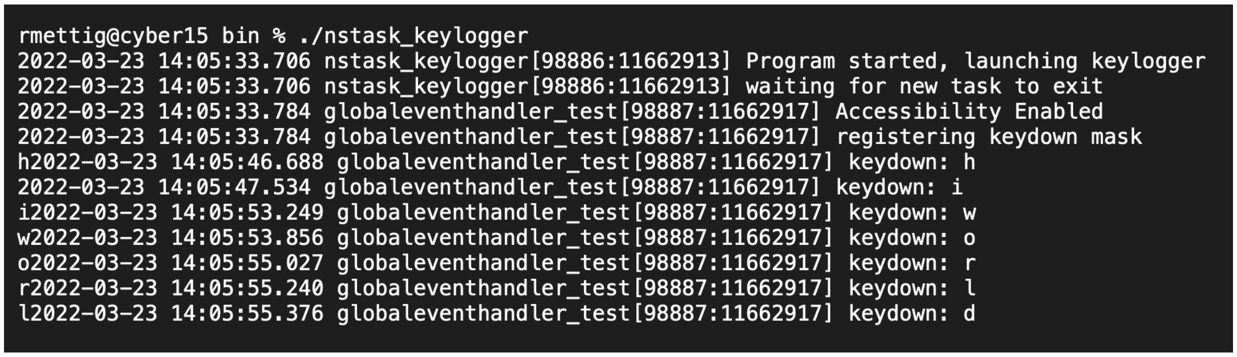
```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
-f $F --shift=568819712 --dtb=34499395584 mac_procdump --dump-dir=out --pid=20500
Volatility Foundation Volatility Framework 2.6.1
Task                     Pid    Address            Path
------------------------ ------ ------------------ ----
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

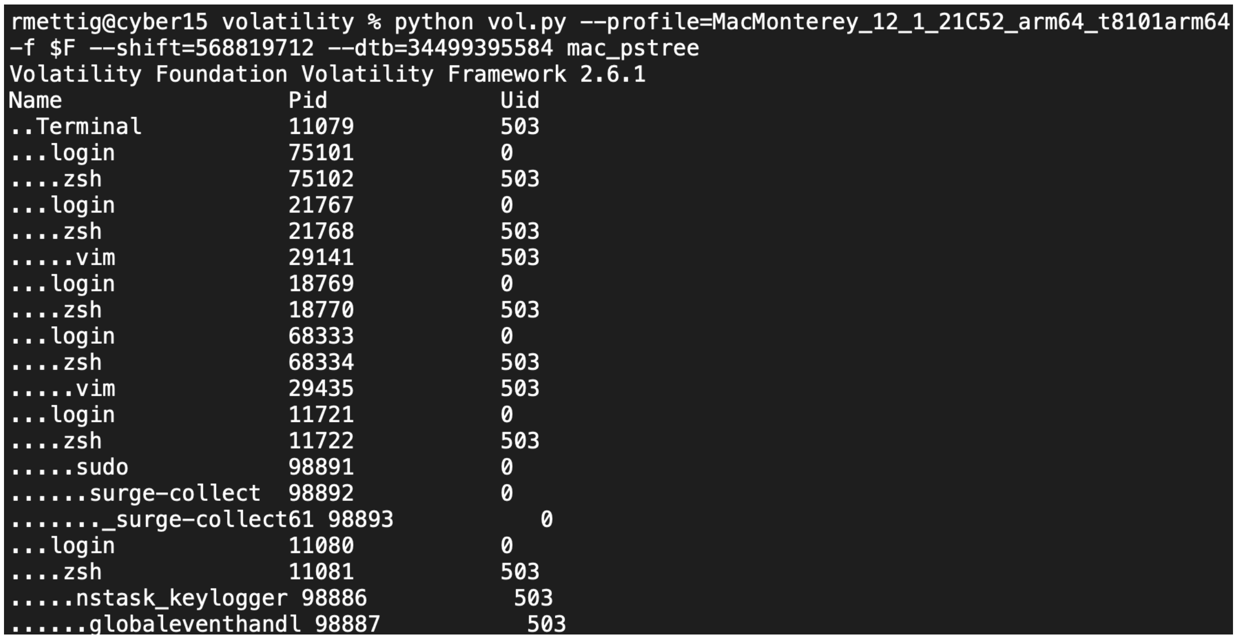
After Rosetta 2 is first installed, the prompt asking for the user's permission will not show up again the next time we try to run an Intel binary. This is important to note because if Rosetta is already installed in the system then an Intel program can be launched in the background without the user's 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



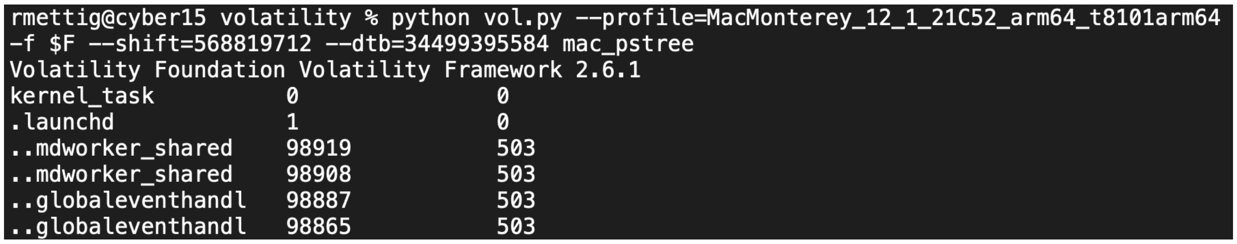
```
rmettig@cyber15 bin % ./nstask_keylogger
2022-03-23 14:05:33.706 nstask_keylogger[98886:11662913] Program started, launching keylogger
2022-03-23 14:05:33.706 nstask_keylogger[98886:11662913] waiting for new task to exit
2022-03-23 14:05:33.784 globaleventhandler_test[98887:11662917] Accessibility 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
```

Figure 4.17. Volatility - mac_pstree output showing keylogger subprocess being spawned

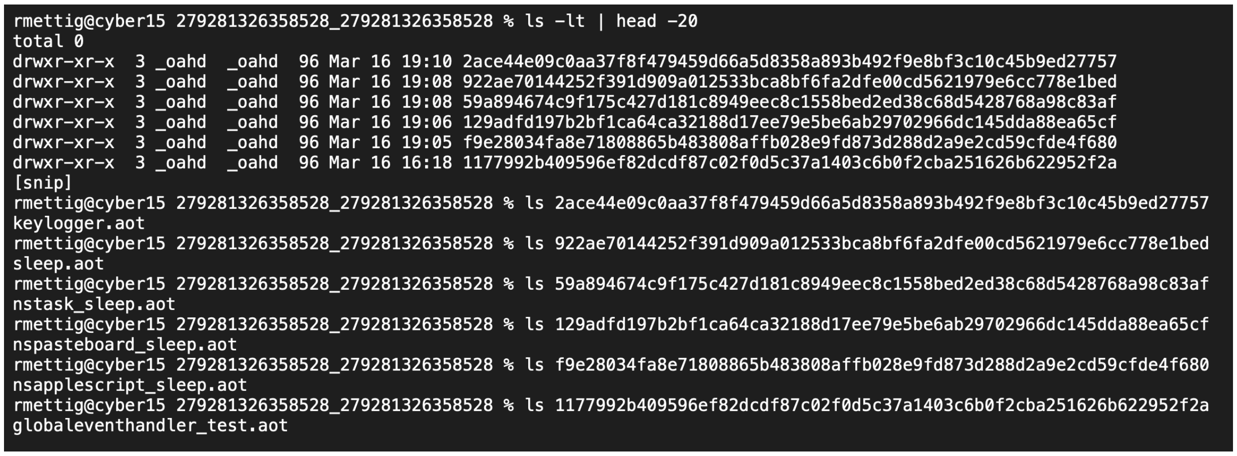


```
rmettig@cyber15 volatility % python vol.py --profile=MacMonterey_12_1_21C52_arm64_t8101arm64
-f $F --shift=568819712 --dtb=34499395584 mac_pstree
Volatility Foundation Volatility Framework 2.6.1
kernel_task          0              0
.launchd             1              0
..mdworker_shared    98919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



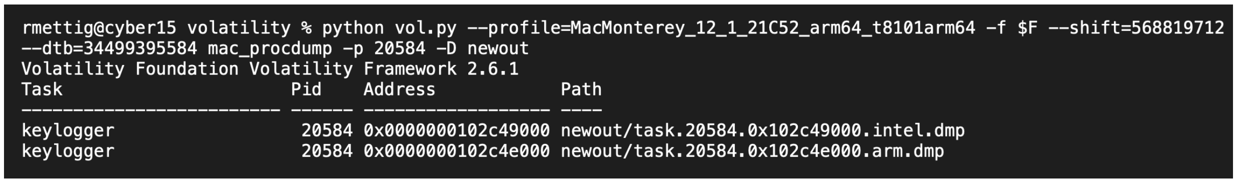
```
rmettig@cyber15 279281326358528_279281326358528 % ls -lt | head -20
total 0
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:10 2ace44e09c0aa37f8f479459d66a5d8358a893b492f9e8bf3c10c45b9ed27757
drwxr-xr-x  3 _oahd  _oahd  96 Mar 16 19:08 922ae70144252f391d909a012533bca8bf6fa2dfe00cd5621979e6cc778e1bed
drwxr-xr-x  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
```

Figure 4.20. Modification to mac_procdump plugin to dump the original Intel binary and 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. Additional Contributions

It takes a village to get support for a new architecture added to Volatility. The base goal is to ensure that analysts doing memory forensics have the same set of tools available for the M1.

5.1. Parallel Efforts

The research presented so far is one branch of a larger effort being done in collaboration with other researchers from our lab. These contributions include continuous fixes to the Volatility 2 plugins for M1 MacOS, which requires checking for errors or anomalies in the plugin outputs when analyzing samples. We ran all of the plugins that are currently available for Intel mac analysis against an M1 sample to see what broke or not. We found issues such as empty files being carved from memory, files being carved at the wrong address, library and other file paths missing from the output, and much more. So slowly but surely each plugin is getting fixed one at a time. There have also been new plugins developed specifically for the M1, considering the new changes to its environment such as Rosetta 2. On top of that, other efforts analyzed how difficult it would be to run known Intel malware on the M1 and evaluated which security controls would be triggered, how many of them would have to be disabled for the malware to run, and whether it had the intended effects or not. More details on these efforts will be published separately from this paper.

Chapter 6. Conclusion and Future Work

While there were many contributions presented in this work, there is still also much ground to cover both in this area of memory forensics research and documenting M1 internals and the Rosetta 2 runtime.

6.1. Conclusion

Malware threats are becoming more sophisticated by the day, and it is important for analysis tools that are used in the event of an incident be capable of handling a wide variety of platforms. With Apple's introduction of the M1, which is based on the ARM64 architecture, research has to be done in order to better understand how to add support to those architectures for tools like the Volatility Framework. This work aimed to accomplish that in a few different ways. First, by determining any changes to the normal procedure of doing memory forensics, such as understanding which tools are currently supported and how does the environment need to be setup for acquiring memory samples and running Volatility. Second, by testing the Objective-C API for Intel binaries on ARM, which is commonly abused by malware. This part also resulted in a set of curated memory samples that can be used for validation testing Volatility plugins being developed or that need to be fixed. Third, an analysis was done of the Rosetta 2 environment to better understand new potential forensic artifacts that can be extracted through memory analysis. Lastly, the shared efforts introduced multiple fixes to Volatility 2 plugins to add support to the M1 while examining the possibility of preexisting macOS malware being able to run on the M1 because of Rosetta.

6.2. Future Work

Like previously mentioned, there is still much work to be done. So far most of the research presented in this work aims to understand the changes introduced by the M1. Even though there are a few new Volatility plugins such as `mac_rosetta` and new updates introduced to existing plugins such as `mac_procdump` that are now able to dump both the Intel binaries and their respective AOT files on the M1, most of the remaining mac plugins have yet to be updated to support the new architecture. Lastly, there is still much unknown about the Rosetta 2 runtime and how it could be targeted as an attack surface or what other kinds of forensic artifacts can be recovered from it.

References

[1] [REF:https://kb.parallels.com/125561], 20212, title = Install macOS Monterey 12 virtual machine on a Mac with Apple M1 chips, author = Paralles Knowledge Base.

[2] Lawrence Abrams. (2018) Clipboard hijacker malware monitors 2.3 million bitcoin addresses. [Online]. Available: https://www.bleepingcomputer.com/news/security/clipboard-hijacker-malware-monitors-23-million-bitcoin-addresses/

[3] Stephanie Archibald. (2017) Running executables on macos from memory. [Online]. Available: https://blogs.blackberry.com/en/2017/02/running-executables-on-macos-from-memory

[4] @08Tc3wBB, "Kernel exploitation on apple's m1 chip," https://objectivebythesea.com/v4/talks/OBTS_v4_08tc3wbb.pdf, 2021.

[5] Apple. (2006) Apple to use intel microprocessors beginning in 2006. [Online]. Available: https://www.apple.com/newsroom/2005/06/06Apple-to-Use-Intel-Microprocessors-Beginning-in-2006/

[6] Apple. (2016) Applescript language guide. [Online]. Available: https://developer.apple.com/library/archive/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html

[7] Apple. (2019) About system integrity protection on your mac. [Online]. Available: https://support.apple.com/en-us/HT204899

[8] Apple. (2020) Apple unleashes m1. [Online]. Available: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[9] Apple. (2021) Introducing m1 pro and m1 max: the most powerful chips apple has ever built. [Online]. Available: https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built/

[10] Apple. (2022) About the rosetta translation environment. [Online]. Available: https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment

[11] Apple. (2022) Apple unveils m1 ultra, the world's most powerful chip for a personal computer. [Online]. Available: https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/

[12] Apple. (2022) Building a universal macos binary. [Online]. Available: https://developer.apple.com/documentation/apple-silicon/building-a-universal-macos-binary

[13] Apple. (2022) Installing a custom kernel extension. [Online]. Available: https://developer.apple.com/documentation/apple-silicon/installing-a-custom-kernel-extension

[14] Apple. (2022) Writing arm64 code for apple platforms. [Online]. Available: https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms

[15] Apple Platform Security. (2021) Rosetta 2 on a mac with apple silicon. [Online]. Available: https://support.apple.com/guide/security/rosetta-2-on-a-mac-with-apple-silicon-secebb113be1/web

[16] A. Case and G. G. Richard III, "Detecting objective-c malware through memory forensics," *Digital Investigation*, vol. 18, pp. S3–S10, 2016.

[17] J. Duke, "Memory forensics comparison of apple m1 and intel architecture using volatility framework," 2021.

[18] V. Foundation, "VMware Snapshot File," https://github.com/volatilityfoundation/volatility/wiki/VMware-Snapshot-File, 2014.

[19] Kaspersky. (2015) Duqu 2.0 - technical details. [Online]. Available: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07205202/The_Mystery_of_Duqu_2_0_a_sophisticated_cyberespionage_actor_returns.pdf

[20] Koh M. Nakagawa. (2021) Reverse-engineering rosetta 2 part1: Analyzing aot files and the rosetta 2 runtime. [Online]. Available: https://ffri.github.io/ProjectChampollion/part1/

[21] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* John Wiley & Sons, 2014.

[22] M. Manna, A. Case, A. Ali-Gombe, and G. G. Richard III, "Modern macos userland runtime analysis," *Forensic Science International: Digital Investigation*, vol. 38, p. 301221, 2021.

[23] MITRE. (2021) Cobalt strike. [Online]. Available: https://attack.mitre.org/software/S0154/

[24] National Security Agency. (2022) Ghidra software reverse engineering framework. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[25] Patrick Wardle. (2017) Osx/proton.b - a brief analysis, at 6 miles up. [Online]. Available: https://objective-see.com/blog/blog_0x1F.html

[26] M. Roy, "Announcement: Vmware fusion for apple silicon public tech preview now available," 2021. [Online]. Available: https://blogs.vmware.com/teamfusion/2021/09/fusion-for-m1-public-tech-preview-now-available.html

[27] C. Scarborough, "macos keylogger," https://github.com/caseyscarborough/keylogger, 2021.

[28] S. Skowronek, "Reverse engineering the m1," https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Reverse-Engineering-The-M1.pdf, 2021.

[29] Volexity. (2021) Surge. [Online]. Available: https://www.volexity.com/products-overview/surge/

[30] T. Zidenberg, "tsahee arm64," https://github.com/tsahee/volatility/tree/arm64, 2022.

Vita

Raphaela Santos Mettig Rocha, born in Salvador, Bahia, Brazil, received her bachelor's degree from Louisiana State University in May 2020. She has been with the LSU Applied Cybersecurity Lab (ACL) since her sophomore year where she has had the opportunity to explore her interests in the cybersecurity field and been contributing to research work there since then. She plans to receive her master's this May 2022 and continue working in cybersecurity.