9-1-2023

Assessing the threat of Rosetta 2 on Apple Silicon devices

Raphaela Mettig
Louisiana State University

Charles Glass
Louisiana State University

Andrew Case
Volatility Foundation

Golden G. Richard
Louisiana State University

Recommended Citation

Mettig, R., Glass, C., Case, A., & Richard, G. (2023). Assessing the threat of Rosetta 2 on Apple Silicon devices. *Forensic Science International: Digital Investigation, 46* https://doi.org/10.1016/j.fsidi.2023.301618

Forensic Science International: Digital Investigation 46 (2023) 301618

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

Assessing the threat of Rosetta 2 on Apple Silicon devices

Raphaela Mettig [b,*], Charles Glass [b,c], Andrew Case [a,b], Golden G. Richard III [b,c]

[a] Volatility Foundation, United States of America
[b] Center for Computation and Technology, Louisiana State University, United States of America
[c] School of Electrical Engineering & Computer Science, Louisiana State University, United States of America

ARTICLE INFO

Keywords:
Rosetta 2
Apple Silicon analysis
ARM64 analysis
Malware analysis
Digital forensics

ABSTRACT

In November 2020, Apple introduced a new architecture, Apple Silicon, that would power all new laptops and desktops. Based on ARM64 and with many custom features added by Apple, this marked a complete switch from the Intel-based systems that have powered Apple laptops and desktops for many years. With such a radical change, it was obvious that many existing digital forensics and incident response techniques would need to be re-evaluated on the platform. Similarly, several new additions to the operating system are interesting as potential abuse vectors for malware and malicious actors. In this paper, we document our effort to understand the largest threat surface unique to Apple Silicon devices. This feature, called Rosetta 2, allows 64-bit Intel applications and libraries to execute seamlessly on Apple Silicon. Rosetta 2 achieves this by translating Intel-specific code on the fly into functionally equivalent Apple Silicon instruction sequences. Through this feature, a significant number of existing applications can be executed without needing to be recompiled for Apple Silicon. Apple added this capability to ease the transition to Apple Silicon devices as not all legacy applications will be recompiled in a timely manner and some may not be ported at all. Such a capability piqued our research team's interest as there have been many malware samples created for Intel systems that were used to target individuals, corporations, and governments. Our goal was to discover which classes of existing malware samples and which offensive techniques remain functional via Rosetta 2. To accomplish this, we acquired a wide ranging set of macOS malware samples, executed them on Apple Silicon devices, and observed the results. We then documented the APIs abused by existing macOS malware and wrote proof-of-concept applications that mirrored these techniques. In this paper, we document the results of this research, including a discussion of the ease with which existing malicious applications and techniques seamlessly function through Rosetta 2.

1. Introduction

Given the sophistication and dangers of modern malware, it is necessary that high-risk individuals, private businesses, government organizations, and military entities have the ability to detect and prevent targeted threats. For this to be possible, it is necessary for the information security community to stay ahead of attackers and understand potential abuses of widely used platforms. Without this ability, attackers may successfully compromise networks for long periods of time before detection, leading to loss of intellectual property, government secrets, and potentially the life and/or freedom of high-risk individuals.

In this paper, we present a deep study of Rosetta 2, which is a unique feature on Apple's new computer architecture and operating system versions. We studied Rosetta 2 because it introduces a large, new attack surface for Apple devices utilizing the new Apple Silicon architecture. Announced in November 2020, the Apple Silicon architecture is a complete break by Apple from producing Intel-based desktops and laptops. The new architecture is based on ARM64 and contains a number of exclusive features that are not covered by existing information security research. This gap potentially leaves users vulnerable as threats will go unnoticed until digital forensics tools can successfully detect and analyze the areas that malware can abuse.

To work towards closing this gap, we present our analysis of Rosetta 2 and the potential abuses that could be perpetrated by malicious actors. Rosetta 2 allows applications compiled for 64-bit Intel systems to run directly on Apple Silicon devices. This is accomplished by Rosetta 2 translating the code contained within the Intel executables to semantically-equivalent Apple Silicon instructions. Once converted, the Intel applications and libraries can, with very few exceptions, run as if they were executing on an Intel-based system. One notable excep-

* Corresponding author.
E-mail addresses: rsanto2@lsu.edu (R. Mettig), cglass5@lsu.edu (C. Glass), andrew@dfir.org (A. Case), golden@cct.lsu.edu (G.G. Richard).

https://doi.org/10.1016/j.fsidi.2023.301618
Received 12 December 2022; Received in revised form 14 July 2023; Accepted 12 August 2023
Available online 7 September 2023

tion is that Intel-based virtualization products will not function using Rosetta 2. While Rosetta 2 eases the transition for users moving to Apple Silicon devices, it also would seemingly allow Intel-based macOS malware to infect the new devices. We also note that mixing Intel and ARM code within the same process' memory has interesting implications for memory forensics and malware analysis research on the platform.

To understand the threats posed by Intel-malware executed on Apple Silicon devices, we developed a two part research plan. The first part of our plan was to gather historically dangerous macOS malware samples that were used in real attack campaigns by advanced persistent threat (APT) groups. This type of malware is often the most stealthy and sophisticated as it is used in attacks against targets that potentially employ a high level of operational security. Given that they were used to target vulnerable individuals and corporate settings, we chose these samples for our analysis. Once we gathered these samples, we then executed them on Apple Silicon devices to determine which malicious features successfully worked through Rosetta 2. This was aimed at informing us of the real world threats posed by the dozens of powerful malware samples historically targeted at macOS systems.

The second part of our research was to study and document the APIs most often abused by macOS malware to monitor users and user activity. We then developed proof-of-concept (POC) applications that mimicked the abuse of these APIs to determine how Rosetta 2 reacted to them being utilized. Interestingly, we determined that unsigned Intel applications can be executed on Apple Silicon devices if a user is tricked into allowing them whereas there is no mechanism for unsigned ARM64 applications to execute. This seemingly would push malware authors to compile their code for Intel instead of ARM64, and represents an unintended bypass capability for macOS malware.

We believe the results of our research represent a strong indicator of the need for deep analysis of Apple Silicon devices as well as the development of digital forensics techniques that can detect new methods for system access and abuse on Apple Silicon.

2. Related work

2.1. Apple Silicon analysis

Given its relatively new entrance to the market, there have only been a few deeply technical analyses of the Apple Silicon architecture. At Black Hat 2021, the Corellium team presented a deep dive into their effort to reverse engineer Apple Silicon devices at the hardware level (Skowronek, 2021). At the Mac security conference Objective By the Sea, the ZecOps team presented methods for kernel exploitation of Apple Silicon devices (@08Tc3wBB, 2021). A report by Patrick Wardle was also published in 2021 on reverse-engineering the first native M1 malware (Wardle, 2021c), and in 2022 Wardle also published a book focused on macOS malware analysis (Wardle, 2022). The "ARM Assembly Internals and Reverse Engineering" book recently published Maria Markstedter covers some macOS malware reverse engineering for Apple Silicon (Markstedter, 2023). Most closely related to our work is Project Champollion and its associated blog posts (Nakagawa, 2021). This research effort details debugging and examination of internal structures of the Rosetta 2 emulator as well as the structure of the produced ahead-of-time (AOT) files.

3. Testing historical malware through Rosetta 2

There are a significant number of malware samples that have been created over the years for Intel-based Apple systems. One aspect of our research was to investigate if any of these samples would function as intended through Rosetta 2. The results of this work would also help to inform the threat landscape of Apple Silicon devices.

Table 1
Summary of environments used for testing.

Environment	Bare metal/VM	macOS version	Build
Malware Testing (Host)	Bare Metal	12.2.1	21D62
Malware Testing (VM)	VM	12.2	21D49
EvilQuest Malware	Bare Metal	12.2.1	21D62
API Testing	Bare Metal	12.1.0	21C52

3.1. macOS security mechanisms

Before executing existing malware, we first studied existing security mechanisms for Apple Silicon devices to understand how they may interfere with malware operations.

3.1.1. File quarantine

File quarantine is a security feature where files downloaded from the internet are marked as quarantined. When quarantined files are first run, a user needs to give permission via a dialog box before the file is allowed to execute. Files downloaded via *curl* and some other methods, however, are *not* marked as quarantined (Owens, 2021), so this method of protecting users is capable of being bypassed.

3.1.2. Gatekeeper

Gatekeeper is a security feature that is intended to prevent apps developed by an unknown or invalid developer from running without user consent. This method is potentially more effective than file quarantine, but is still susceptible to being overridden for individual apps or having system settings updated to allow apps from anywhere to run. In addition, malware authors can bypass this mechanism by signing their apps with illegitimately obtained but valid developer IDs (Wardle, 2021a).

3.1.3. Notarization

Notarization is a security feature that automatically scans quarantined apps for issues related to code-signing or malicious content (Apple Documentation, 2023). If macOS encounters a previously unnotarized program, it will inform the user and then let them bypass and run it if they choose (hoakley, 2020).

3.1.4. XProtect

As a final security measure, Apple also develops an integrated antivirus called *XProtect* that uses YARA signatures to detect malware and prevent it from running (Apple Platform Security Guide, 2022). When an executable is flagged as malware, running the executable requires either overriding malware protection for that app or disabling System Integrity Protection (SIP).

3.2. Environment setup

Excluding one sample, *EvilQuest*, the malware samples tested for this portion of the research were executed on virtual machines (VMs) running on Parallels Desktop Pro 17.1.1-51537. The operating system of the VMs was macOS Monterey 12.2, build 21D49, and the operating system of the host Mac was macOS Monterey 12.2.1, build 21D62. The host that the memory was acquired from was running macOS Monterey 12.1.0, build 21C52. Table 1 summarizes all test environments. All bare metal systems and VMs were using the Apple Silicon architecture.

Rosetta 2 is not installed by default, but is instead installed upon the first execution of an Intel-compiled application. We ensured that this process happened on each VM prior to execution of malware. Command Line Developer Tools were installed as well, as some malware utilized functionality from tools in this suite. On the VMs, Wireshark and various built-in command line tools such as *ps* were used to monitor high-level functionality of the malware. Each VM was within what Parallels calls a *Host-Only Network* (Parallels, 2020), where the VM is in a separate subnet that can only connect to a gateway and other VMs within the subnet.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www. apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.UserAgent.va</string>
    <key>LimitLoadToSessionType</key>
    <string>Aqua</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Users/$USER/Library/Preferences/UserAgent/lib/UserAgent</string>
        <string>--runMode</string>
        <string>ifneeded</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>StartInterval</key>
    <integer>600</integer>
    <key>ThrottleInterval</key>
    <integer>2</integer>
    <key>WorkingDirectory</key>
    <string>/Users/$USER/Library/Preferences/UserAgent/lib</string>
</dict>
</plist>
```

Fig. 1. *CDDS* ensuring persistence through a Launch Agent.

```
$ ps aux | grep UserAgent
USER   PID  %CPU  %MEM  VSZ       RSS  TT STAT STARTED TIME     COMMAND
$USER  485  0.6   0.1   34727104  6804 ?? S    3:38PM  0:04.30  /Users/$USER/Library/Preferences/UserAgent/lib/UserAgent -runMode ifneeded
$USER  628  0.4   0.1   34618864  8680 ?? S    3:39PM  0:00.50  kAgent /Users/$USER/Library/Preferences/UserAgent/lib/data2 485
```

Fig. 2. *UserAgent* running via a Launch Agent with arguments specified.

Each VM was also set up such that it was the DNS Server under its networking configuration. This setup allowed the observation of DNS queries made during dynamic analysis of the malware. Since there is currently no snapshot functionality on Apple Silicon images for Parallels, multiple samples were run on one VM before preparing and using another VM for more samples.

For all VMs, System Integrity Protection (SIP) remained enabled. As a result, a few samples identified by macOS as malware were unable to run on the VMs. Most users will not disable SIP, so this is a reasonable setup. Furthermore, it was not possible to disable SIP in VMs running under the version of Parallels that we used, because Recovery Mode for macOS is not currently supported. Individual applications identified as malware could still be executed by overriding malware protection, but executables, disk images, and package installers could not.

Three malware samples were also run directly on physical hardware (as opposed to in a VM) while the machine was isolated from the internet, so that behavior could be observed. The physical machine was wiped and rebuilt after each sample was executed and a memory image was acquired. The use of a bare metal system mirrored the environment of the majority of infections in the wild.

3.3. Functional malware samples tested

3.3.1. CDDS/MacMa

CDDS/MacMa was discovered in November 2021 and targeted visitors to pro-democracy websites in Hong Kong. It has numerous capabilities, including execution of terminal commands submitted remotely, file transfer, audio recording, and more (Hernandez, 2021).

After running the 2021 version of the installer, a payload was dropped that triggers warnings such as attempting to control the computer using accessibility features and accessing the microphone. *UserAgent* is the executable that is dropped and persists via a Launch Agent. Launch Agents are perhaps the most common method for malware to persist on macOS. Launch Agents can be used to take actions whenever directories change, start jobs at set intervals, and even run scripts every time a user logs in Applelaunch (2016). Launch Daemons are largely equivalent, except they run on behalf of the root user or another specified user (launchd, 2022).

As an example of a Launch Agent, *CDDS*'s can be seen in Fig. 1. *UserAgent* running accordingly is shown in Fig. 2.

3.3.2. DazzleSpy

DazzleSpy was discovered in January 2022 and was distributed via a compromised Hong Kong pro-democracy website. It supports fully interactive RDP sessions, the ability to dump the Keychain on vulnerable systems, and self-deletion (Léveillé and Cherepanov, 2022). This sample comes from a similar campaign to *CDDS*, with the main difference being the payloads (*CDDS* and *DazzleSpy*) that are delivered.

After running *DazzleSpy*, connection attempts to its probable command and control (C&C) were observed. In addition, a Launch Agent was loaded that will launch the executable each time the user logs in.

3.3.3. EvilQuest

EvilQuest was discovered in June 2020 and was distributed predominately through pirated programs. It has the ability to ransom a host, infect files to run its payload, search for and kill processes, and harvest data (Wardle, 2020).

The malware bundled with the pirated program was identified as such by XProtect, so it was prevented from running. As discussed in prior sections, it is not possible to fully disable malware protections for executables with the current version of Parallels for Apple Silicon Mac VMs, because Recovery Mode is not supported and entering Recovery Mode is necessary to disable SIP. As this sample was of particular interest due to its distribution vector, capabilities, and relative recency, it was selected for further analysis and run on bare hardware.

From initial observation, persistence was set, test binaries were infected, and a fake process named *Avast* was killed. The file infection was marked by a terminating hex sequence *0xdeadface* and maintains the normal functionality of the infected binary. Networking could not be tested because it was disabled on the host to prevent damage to

```
$ ps aux | grep silent
USER  PID    %CPU  %MEM   VSZ        RSS   TT   STAT  STARTED  TIME     COMMAND
root  1529   0.1   0.0    408666384  7792  ??   Ss    6:38PM   0:00.02  sudo /Library/AppQuest/com.apple.questd --silent
root  1530   0.1   0.0    34260360   2668  ??   S     6:38PM   0:00.01  /Library/AppQuest/com.apple.questd --silent

$ hexdump ~/Downloads/testBinary | tail -n 10
0021950 65 61 64 65 72 00 5f 68 65 6c 6c 6f 00 5f 6d 61
0021960 69 6e 00 5f 6e 75 6d 00 5f 6e 75 6d 32 06 5f 6e
0021970 75 6d 62 65 72 5f 00 5f 5f 5f 73 74 72 63 70 79
0021980 5f 63 68 6b 00 5f 66 72 65 65 00 5f 67 65 74 63
0021990 68 61 72 00 5f 6d 61 6c 6c 6f 63 00 5f 70 72 69
00219a0 6e 74 66 00 64 79 6c 64 5f 73 74 75 62 5f 62 69
00219b0 6e 64 65 72 00 5f 5f 64 79 6c 64 5f 70 72 69 76
00219c0 61 74 65 00 00 00 00 00 03 70 57 01 00 ce fa ad
00219d0 de
```

Fig. 3. *EvilQuest* file infection and persistence.

other machines in our environment, but based on other functionality it is likely to have attempted to connect to C&C.

Fig. 3 shows both a running Launch Agent and file infected from *EvilQuest*.

3.3.4. FinSpy

The macOS version of *FinSpy* was discovered by Amnesty International in Fall 2019. The original Windows version was distributed through a backdoored version of Adobe update (Amnesty International, 2020) and the macOS version was distributed as a trojaned application with a Turkish name (osxreverser, 2020). It is used as spyware.

After the FinSpy disk image was mounted and the installer app was run, a malicious installer was dropped and executed at the same time a network request to download a legitimate version of Adobe AIR was made. The malicious installer then detected it was running inside virtualization software and stopped running. As the focus of this portion of the research was confirming high-level functionality through Rosetta 2 translation, using a debugger to get around this check was outside the scope of our research. It appears likely that the malware is fully functional when executed under Rosetta.

3.3.5. Hydromac

Hydromac was discovered in the middle of 2021 and was distributed via the *Tarmac* and *Bundlore* malware toolsets (lordx64, 2021). It is thought to be a downloader/stager, as it has the ability to check for installed anti-virus programs and can download and execute programs.

After giving execution permissions, it runs and attempts to connect out to C&C. No other functionality was observed.

3.3.6. Komplex

Komplex was discovered in late 2016 and was associated with APT28 Sofacy. It likely targeted individuals working in aerospace and was distributed as an attachment in an email. It has functionality including remote command execution and information collection about victim systems (Creus et al., 2016).

Upon running *Komplex*, a PDF in Russian opened that appeared to relate to a space program. It also dropped and ran an executable that checked for connectivity, as well as creating a Launch Agent that launches that script at startup. It also deleted itself after dropping the other executable. However, an additional script that was supposed to be responsible for loading the Launch Agent did not get created or run, so persistence was ultimately not successful. This was confirmed by rebooting the system and noting that the payload did not start running. The VM this malware ran on was isolated from the internet, so the connectivity check failed.

Parallels Host-Only mode did not seem to allow a VM to see other VMs on the same subnet, though the documentation implies it should. As a result, using a program like INetSim (Inetsim, 2021) on another VM to return 200's to the malware did not seem possible and was out of scope since functionality was already observed.

3.3.7. MacDownloader

MacDownloader was discovered in early 2017 and was associated with the Iranian APTs Charming Kitten and Flying Kitten. It was distributed as both an Adobe Flash installer and BitDefender adware remover, and has the ability to harvest and upload sensitive information (IranThreats, 2017).

This malware completes a fake installation while asking the user for their credentials. A file is created in a temp directory that is populated with sensitive information, including username and password, Keychain files, running processes, and more. Traffic to its known C&C domain was not observed in our testing.

In Fig. 4, the harvested info from a *MacDownloader* execution can be seen.

3.4. Realtime-spy

This *Realtime-Spy* sample is from early 2018, was distributed via the *realtime-spy-mac[.]com* website, and has the ability to upload screen recordings, keystrokes, and other user activity it has collected (Hyvärinen, 2018).

While the sample runs, it can be seen reaching out to its namesake domain and attempting to record the Mac's screen. See Fig. 5.

3.4.1. Ventir

Ventir was discovered back in 2014 and has the ability to execute commands, exfiltrate files, and record keystrokes on a fully infected system (Kuzin, 2014).

It dropped 5 binaries and then removed itself, created and loaded a launch agent for persistence, and ran one of the dropped binaries. However, the keylogging functionality fails as the binary responsible is a universal binary compiled for i386 and PowerPC. Rosetta 2 does not translate these architectures and macOS no longer supports 32-bit (i386) executables at all.

3.4.2. XCSSET

XCSSET was discovered in early or mid 2020, spreads through infected XCode projects, and attempts to steal credentials and data through trojanized browsers (Long, 2020).

Running either of the associated scripts fails, as they are fingerprinted as malware and prevented from executing. Running the infected XCode application after overriding malware protection, however, results in a connection attempt to C&C infrastructure. None of the other infection markers, such as processes being killed or fake apps being created, are observed due to the failed connection attempt.

3.4.3. ZuRu

ZuRu was discovered in September 2021. It spread through the search engine Baidu, where the authors paid to show at the top of the search results for "iTerm2" via a sponsored link. As a result, it appeared above even the legitimate iTerm2, likely tricking many users. Its functionality includes downloading and executing additional files that exfiltrate information and leverage remote access (Wardle, 2021b).

```
[

"OS version: Darwin user-Mac.local 21.3.0 Darwin Kernel Version 21.3.0: Wed Jan 5 21:37:58 PST 2022; root: xnu-8019.8@.24~20/
RELEASE_ARM64_VMAPPLE x86_64",

"Root Username: "$USER"",

"Root Password: "password"",

"Keychains loaded in current user <NSAppleEventDescriptor: 'utxt'(" A"/Users/$USER/Library/Keychains/login. keychain-db"r "/
Library/Keychains/System.keychain"")>",

"Local ip address: [snip]",

"ifconfig: [snip]",

[

"Applications/Utils/",
"Applications/Safari.app/",
"Applications/Utilities/",
"Applications/Wireshark.app/",
"Applications/Mixed%20In%2@Key%208.app/"
J,
[
"process name is: loginwindowt PID: 140 Run from: file:///System/Library/CoreServices/.../loginwindow",
"process name is: ViewBridgeAuxiliaryt PID: 229 Run from: file:///System/Library/PrivateFrameworks/.../ViewBridgeAuxiliary",
"process name is: universalaccessdt PID: 355 Run from: file:///usr/sbin/universalaccessd",
"process name is: Terminalt PID: 367 Run from: file:///System/Applications/Utilities/.../Terminal",
"process name is: Notification Centert PID: 386 Run from: file:///System/Library/CoreServices/.../NotificationCenter",
"process name is: AppSSOAgentt PID: 398 Run from: file:///System/Library/PrivateFrameworks/.../AppSSOAgent",
"process name is: ViewBridgeAuxiliaryt PID: 4@7 Run from: file:///System/Library/PrivateFrameworks/.../ViewBridgeAuxiliary",
_ "process name is: talagentt PID: 488 Run from: file:///System/Library/CoreServices/talagent",

[snip]
```

Fig. 4. Results of *MacDownloader*'s harvested information about OS, user, IP, applications, processes, etc.

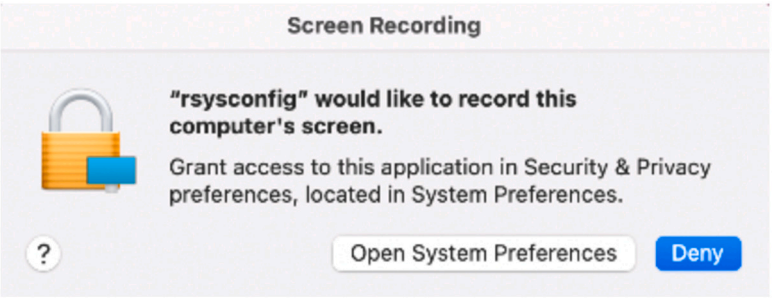

Fig. 5. *Realtime-Spy* attempting to screen record.

An initial run was terminated at launch because it was not able to load in a custom malicious dynamically linked library. After investigation, it was determined that the library wasn't loaded due to macOS expecting it to be ARM64 rather than the x86_64 library it found. The library was expected to be ARM64 because this application was a universal binary, so by default it ran as ARM64 on an M1 Mac. After specifying the code slice to run by selecting "Open using Rosetta" or running it in Terminal with *arch -x86_64*, it did not crash. It attempted to connect out to C&C but stopped running because it did not receive a response.

This successful execution means that Rosetta 2 handles both translation of x86_64 code and also any x86_64 dependencies, as expected. However, when an app is started as ARM64, even if a translatable x86_64 dependency is found, the app will fail to run. The implication then is that if a malware author is going to build malware as a universal binary, any dependencies within the app will need to be universal as well. This implication is confirmed through Apple documentation. Otherwise, their app will fail on Apple Silicon Macs, since ARM64 is the default architecture that will run in a universal binary (Unix, 2010). The other way to prevent a crash would be for the author to force execution as x86_64 on an infected host, but at that point there would be no reason to have bothered additionally building as ARM64 in the first place.

In Fig. 6, *ZuRu*'s crash report when run as ARM64 can be seen. It details errors regarding incompatible architecture due to finding an x86_64 dependency that was expected to be ARM64. This crash in particular is notable, because it is an example of malware that would have worked by default if it wasn't modernized to a universal binary. But because the authors of ZuRu did attempt to modernize it, their malware crashes on every Apple Silicon machine when the default code slice is run. If they had left it to Rosetta 2 to handle their execution, this would not have been the case.

3.5. Non-functional malware samples tested

Malware was determined to be nonfunctional on Apple Silicon devices if it needed significant modification or to be handheld through bypassing multiple security mechanisms for different stages of its payloads. As an example, *Applejeus* was caught by Gatekeeper, then had a second and third stage get immediately caught by malware protection. Another example, *SysJoker*, needed to have its file extension modified on top of being provided executable permissions to run correctly. This modification changed the entire attack vector of *SysJoker*, as its file extension is thought to play an important part of the initial infection vector, either as a malicious media player attachment or as a TypeScript file inside of an infected package.

```
Crashed Thread: 0

Exception Type: EXC_CRASH (SIGABRT)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Exception Note: EXC_CORPSE_NOTIFY

Termination Reason: Namespace DYLD, Code 1 Library missing
Library not loaded: @executable_path/../Framewrks/libcrypto.2.dylib
Referenced from: /Users/USER/Downloads/*/iTerm.app/Contents/MacOS/iTerm2

Reason: tried: '~/Downloads/ZuRu/iterm/iTerm.app/Contents/MacOS/../Frameworks/
libcrypto.2.dylib' (mach-o file, but is an incompatible architecture (have 'x86_64',
need 'arm64e')), '/usr/IocaI/lib/libcrypto.2.dyIib'  (mach-o file, but is an
incompatible architecture (have 'x86_64', need 'arm64e'), '/usr/lib/
libcrypto.2.dylb' (no such file)
(terminated at launch; ignore backtrace)

[snip]
```

Fig. 6. *ZuRu* crashing due to an incompatible dependency.

```
$ file Crisis
Crisis: Mach-O executable i386
$ ./Crisis
zsh: bad CPU type in executable: ./Crisis
```

Fig. 7. *Crisis* incompatibility.

3.5.1. Applejeus/MacLoader

Applejeus was discovered around the middle of 2018, is associated with a North Korean APT, was distributed through phishing emails and trojanized programs (Global Research & Analysis Team and Kaspersky Lab., 2018), and has functionality including execution of memory-only payloads.

The initial package installer ran, but did not function as intended. It gave root permissions to the malicious executable *unioncryptoupdater* and moved it to a separate location, but that executable was not launched. This failure seems to be due to *unioncryptoupdater* being identified as malware (MACOS.8d038b3) and prevented from running. The same occurs with the *CrashReporter* malware that another *Applejeus* package installer tries to drop and run as a Launch Daemon.

3.5.2. Crisis

Crisis was discovered in 2016, appeared to be sold as a part of government surveillance tools, and used Adobe-related phishing as an infection method. It spies on users via audio recording, stealing browser data, and more (Myers, 2012).

Crisis failed to run due to being a 32-bit (i386) executable, which is incompatible with Rosetta 2 (and modern versions of macOS in general).

In Fig. 7, Rosetta being unable to translate the *Crisis* i386 executable can be seen.

3.5.3. Shlayer

Shlayer was originally discovered in 2018, with new variants found through 2021. It is distributed through malicious sites intended to be discovered through search engine optimization, that then redirect to fake Adobe Flash installation pages. A recent variant from last year had the ability to run only with a user double-click, as it utilized 0-days to bypass most Apple security mechanisms. After infection, it only attempts to download adware (Bradley, 2021).

The disk image that contains the installer, *AdobeFlashPlayer.dmg*, was flagged as malicious and wasn't mounted. The installer package within, Player.pkg, was not flagged as malware and ran, but didn't appear functional at all. This lack of functionality is unsurprising because the malware relied on 0-days that were patched on the OS build it was run on. The ways that the macOS security mechanisms were bypassed were not highly technical, however. They essentially were the result of leaving out an *Info.plist* file of an app bundle that was script-based.

3.5.4. Silver Sparrow

Background

Silver Sparrow was discovered in February 2021, is distributed in installer packages, is a universal binary, and has an unknown final payload. The widest held opinion is that the second stage installs adware, based on similarities between data from the C&C that includes the arg *upbuchupsf*, and an affiliate code often used by adware. It also has roughly 30,000 detections on macOS (Lambert, 2021).

Shortly after running, the executable encounters an error evaluating JavaScript. Two expected files, *version.json* and *version.plist*, are created but they are empty. Two expected shell scripts, *agent.sh* and *very.sh*, are not created. It thus appears to fail. The same JavaScript error occurs when running as both ARM64 and through Rosetta 2 on x86_64, so the failure is apparently not due to a Rosetta 2 translation error.

3.5.5. SysJoker

SysJoker was discovered in December 2021. Examining C&C behavior supports the position that it's used only in targeted attacks. However, it is also possible its extension implies it is TypeScript, to be used as a part of infected software packages. It is a universal binary, and has the ability to execute commands and files based on remote C&C instructions (Mechtinger et al., 2022).

When the malware is executed, the file opens as a playable media file and no infection markers are seen. Persistence is not set, the second binary is not dropped and run, and there is no attempt to connect to its command and control infrastructure. However, after converting it from its extension *.ts* and giving it execute permissions, it is immediately recognized by macOS Finder as a universal Mach-O binary. Running that binary then leads to it functioning correctly. The modified binary was also run through Terminal to force launching as x86_64, and the same functionality was observed. Persistence was set via a Launch Agent that runs the dropped binary, *updateMacOs*, at startup. Attempts to connect to a Google drive to generate its C&C were also observed. Because a signed universal binary is being run on an M1 Mac, the *.ts* version will default to running the ARM64 slice if it runs at all. Therefore, despite the root cause of the inability to function as a *.ts* file not being clear, the problem doesn't appear to be caused by Rosetta 2 translation.

3.6. Testing summary

Tables 2 and 3 show the macOS security mechanisms that were encountered for both the functional and nonfunctional malware. Their columns differ slightly as a result of the functional malware not encountering Notarization and the non-functional malware not encountering File Quarantine.

Table 4 covers the functionality that was observed to determine a successful run. The functionality was grouped into 3 columns: Persistence, Networking, and Harvesting. These correspond to the malware

Table 2
The functional malware and the protections that were overridden.

Malware sample	File quarantine	Gatekeeper	Malware protection
CDDS/MacMa	X		
DazzleSpy	X		
EvilQuest		X	X
FinSpy		X	
Hydromac		X	
Komplex	X		
MacDownloader			X
Realtime-Spy		X	
Ventir		X	
XCSSET			X
ZuRu		X	

Table 3
The non-functional malware and the protections that were overridden.

Malware sample	Gatekeeper	Notarization	Malware protection
AppleJeus/MacLoader	X		X
Crisis	X		
Shlayer		X	X
Silver Sparrow		X	
SysJoker	X		

Table 4
The functional malware and observed high-level behavior.

Malware sample	Persistence	Networking	Harvesting
CDDS/MacMa	X		X
DazzleSpy	X	X	
EvilQuest	X	?	
FinSpy		X	
Hydromac		X	
Komplex		X	
MacDownloader			X
Realtime-Spy		X	X
Ventir	X		
XCSSET		X	
ZuRu		X	

ensuring long-term existence on a system, attempting to connect out from a system, or harvesting information from the system, respectively.

3.7. Discussion

3.7.1. Rosetta 2 translation

From the results we have presented, it is clear that Rosetta 2 is willing to translate x86_64 malware, even in cases where the sample is unsigned, not notarized, and fingerprinted as malware. It does this translation well, as none of the compatible samples malfunctioned as a result of Rosetta 2 failing to translate properly. However, given that the user consistently had to give permissions, and occasionally override malware protections, it is reasonable that it translated the executables. Rosetta 2 is translating what it is told to translate, so it is functioning correctly. That said, because x86_64 malware translated through Rosetta 2 does not have to be signed, while ARM64 malware does and also requires that all dependencies are ported to ARM64, it may be preferable to malware authors to avoid preparing ARM64 versions of their programs at all.

In most cases with new malware tested, and even with much of the older malware tested, overriding malware protection was not necessary because the binaries were not fingerprinted as malware by *XProtect*, despite some being around for over 5 years. As a result, users in most cases would only need to be socially engineered into overriding a warning from File Quarantine or Gatekeeper, at which point they would be infected. Running unsigned code is a vector that would not be possible without Rosetta 2, since ARM64 code requires a valid signature to run at all.

3.7.2. Viability of infection

Most Macs are infected by applications or executables that require user interaction (Wardle, 2021a). Malware authors employ a large number of methods to convince users to infect themselves. These methods include but are not limited to: packaging malware with shareware, bundling malware with pirated applications, masquerading as fake applications, spreading through malicious attachments, and using search engine optimization or sponsored links to have malicious web results ranked as authoritative.

The impact and viability of these samples seems to be lowered due to the general lack of signing and requiring of Rosetta 2 to be installed. However, many users will happily run something anyway when presented with a warning, especially if any of the above methods are used. Indeed, almost all of the tested malware samples utilized one or more of those techniques for initial infection, many to great success.

4. Testing APIs abused by malware

Our second effort for this research project was to determine which, if any, of the APIs historically abused by macOS malware was also available on Apple Silicon devices as well as allowed through Rosetta 2 translation. To begin, we studied and documented the APIs abused and categorized them according their use. Table 5 lists these along with their purpose.

Next, we developed proof-of-concept applications that used these APIs in the same manner as malware has been observed to leverage them. This required a mix of Objective-C and C++ programming. All programs were compiled as 64-bit Intel applications. Each POC application was run in a clean-state copy of a VM. This clean-state includes a reset of permissions for the Terminal application in System Preferences so that we could determine if a particular API caused a system prompt to appear.

4.1. 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 aims to log all of a user's input to a device, usually a keyboard or mouse. In our testing, both methods for performing keylogging led to a prompt asking for Accessibility Permissions to be granted to the Terminal application in order to successfully run. See Fig. 8.

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.

4.1.1. 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 capable of monitoring user keystrokes outside the main keylogger process. Our POC application that used this API ran successfully and was able to capture keystrokes after granting accessibility permissions to the Terminal application.

4.1.2. CGEventTapCreate/CGEventTapEnable

`CGEventTapCreate` and `CGEventTapEnable` are both methods from Apple's CoreGraphics framework. `tapCreate` allows 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 (Scarborough, 2021). The keylogger is written in C and required minor modifications to compile and run. One of the modifications necessary was the location of the log file because the original

Table 5
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

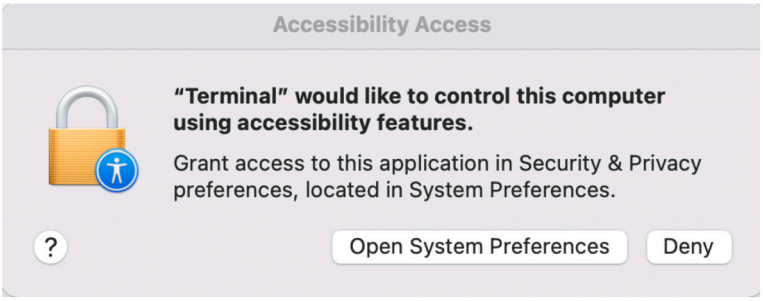

Fig. 8. Terminal prompt requesting Accessibility Permissions.

```
$ ./keylogger-x86_64
ERROR: Unable to open log file. Ensure that you have proper permissions.
```

Fig. 9. Error thrown when the program tried to open the logfile created under SIP protected path.

```
$ ./keylogger
Logging to: ~/Desktop/keystroke.log
hello, this is my keylogger.
^C
$ ./keylogger
Logging to: ~/Desktop/keystroke.log
still logging to the same file.
^C
```

Fig. 10. CGEventTap Keylogger Running.

```
$ cat ~/Desktop/keystroke.log

Keylogging has begun.
Wed Jun 14 15:40:51 2023

hello, this is my keylogger.[return][left-ctrl]c

Keylogging has begun.
Wed Jun 14 15:42:07 2023

still logging to the same file.[return][left-ctrl]c
```

Fig. 11. CGEventTap Keylogger Logfile.

path was at `/var/log/keystroke.log`, and `/var` is one of the directories now protected by SIP. After changing the log file location, the program was able to run without issue. See Figs. 9–11.

4.2. Launching executables

Malware is often programmed to launch executables downloaded from C&C servers or that are dropped to disk during the infection chain. A common technique for malware is to use small loaders as an initial infection vector, which check their running environment for information before deploying full payloads. An example of this is Proton.B malware for macOS X, which uses NSTask Launch to execute a bash process that checks whether its persistence mechanism has been installed and executed (Wardle, 2017).

4.2.1. 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, the operating system will then launch the controlled program. Our POC application uses NSTask to launch `/bin/sleep`, and we verified that the child process of sleep ran without crashing or other issues.

4.2.2. NSAppleScript

NSAppleScript is an Objective-C interface to AppleScript, which is a scripting language developed by Apple that allows the user to directly control and automate scriptable macOS applications by interfacing with application event messages (Apple, 2016). While our POC is much simpler, AppleScript makes it possible to interact with the file system, send text messages through iMessage, launch scripts (including with the `sh` shell), gather volume information, and more. As with, *NSTask::launch*, our POC application ran perfectly through Rosetta 2 and no security mechanisms interfered with the execution.

4.3. 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 (Archibald, 2017).

4.3.1. NSCreateObjectFileImageFromMemory

`NSCreateObjectFileImageFromMemory` is one of the most common ways to load code from memory on a macOS (Archibald, 2017). In this example, the executable opens an existing file on disk, gets its file size and maps it into memory, then `NSCreateObjectFileImageFromMemory` loads it from that mapped

```
$ ./nscreateobjmem
Open file...
Getting file size.
Mapping file in memory...
Done.
Create object in memory...
DEBUG -- loadAddress = 0x10ce36000, st_size = 16480
Program is successfully running in memory only at address 0x10ce36000
```

Fig. 12. Example of NSCreateObjectFileImageFromMemory running in the Terminal with the address.

```
$ ./nspasteboard_sleep
[2023-06-14 15:44:23.751] Writing to clipboard...
[2023-06-14 15:44:23.792] Done.
[2023-06-14 15:44:23.839] Read from clipboard: Hello World
```

Fig. 13. Example of NSPasteboard running in the Terminal.

address. The one restriction encountered was that the program being loaded by `NSCreateObjectFileImageFromMemory` has to be an Apple Bundle, e.g., compiled as a `.app` file. Once our application was packaged as a bundle, we could freely load memory-only libraries. Fig. 12 shows the output generated by running our POC application.

4.4. 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 (Abrams, 2018). By modifying the address, malware can force payments to be sent to a wallet of the malware's choosing.

4.4.1. 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. Registering an NSPasteboard application did not trigger any security prompts.

In Fig. 13, it is shown that the POC application successfully retrieved *Hello World* from the clipboard.

5. Conclusions

In this paper, we have documented the threat posed to new Apple Silicon devices by the corpus of existing malware targeting macOS on the traditional Intel-based platform. Given that both the Intel and Apple Silicon architectures will be in production use within the Apple ecosystem for years to come, it is imperative that information security professionals fully understand the threats on both platforms. By utilizing the information presented in this paper, system administrators and incident response teams can begin to develop approaches to ensuring that Intel malware is not allowed to run undetected within monitored Apple Silicon environments. We also believe there is significant future work that can be done to automatically analyze both the Intel and ARM64 code present when Rosetta 2 is involved with translating and running an Intel-based application. Our team is currently undertaking this task, and we believe the toolsets developed will help to significantly reduce the burden of analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Abrams, Lawrence, 2018. Clipboard hijacker malware monitors 2.3 million bitcoin addresses. https://www.bleepingcomputer.com/news/security/clipboard-hijacker-malware-monitors-23-million-bitcoin-addresses/.

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

Apple, 2016. Applescript language guide. https://developer.apple.com/library/archive/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html.

Apple Documentation, 2023. Notarizing macos software before distribution. https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution.

Apple Platform Security Guide, 2022. Protecting against malware in macos. https://support.apple.com/guide/security/protecting-against-malware-sec469d47bd8/web.

Applelaunch, 2016. Creating launch daemons and agents. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html.

Archibald, Stephanie, 2017. Running executables on macos from memory. https://blogs.blackberry.com/en/2017/02/running-executables-on-macos-from-memory.

Bradley, Jaron, 2021. Shlayer malware abusing gatekeeper bypass on macos. https://www.jamf.com/blog/shlayer-malware-abusing-gatekeeper-bypass-on-macos/.

Creus, Dani, Halfpop, Tyler, Falcone, Robert, 2016. Sofacy's 'komplex' os x trojan. http://researchcenter.paloaltonetworks.com/2016/09/unit42-sofacys-komplex-os-x-trojan/.

Global Research & Analysis Team, Kaspersky Lab., 2018. Operation applejeus: Lazarus hits cryptocurrency exchange with fake installer and macos malware. https://securelist.com/operation-applejeus/87553/.

Hernandez, Erye, 2021. Analyzing a watering hole campaign using macos exploits. https://blog.google/threat-analysis-group/analyzing-watering-hole-campaign-using-macos-exploits/.

hoakley, 2020. How notarization works. https://eclecticlight.co/2020/08/28/how-notarization-works/.

Hyvärinen, Noora, 2018. Spam campaign targets exodus mac users. https://blog.f-secure.com/spam-campaign-targets-exodus-mac-users/.

Inetsim, 2021. Inetsim manpage. https://manpages.debian.org/testing/inetsim/inetsim.1.en.html.

IranThreats. Guarnieri, Claudio, Anderson, Collin, 2017. ikittens: Iranian actor resurfaces with malware for mac (macdownloader). https://iranthreats.github.io/resources/macdownloader-macos-malware/.

Kuzin, Mikhail, 2014. The ventir trojan: assemble your macos spy. https://securelist.com/the-ventir-trojan-assemble-your-macos-spy/67267/.

Lambert, Tony, 2021. Clipping silver Sparrow's wings: outing macos malware before it takes flight. https://redcanary.com/blog/clipping-silver-sparrows-wings/.

launchd, 2022. A launchd tutorial. https://www.launchd.info/.

Léveillé, Marc-Etienne M., Cherepanov, Anton, 2022. Watering hole deploys new macos malware, dazzlespy, in Asia. https://www.welivesecurity.com/2022/01/25/watering-hole-deploys-new-macos-malware-dazzlespy-asia/.

Long, Joshua, 2020. Mac malware exposed: Xcsset, an advanced new threat. https://www.intego.com/mac-security-blog/mac-malware-exposed-xcsset-an-advanced-new-threat/.

lordx64, 2021. Osx/hydromac: a new macos malware leaked from a flashcards app. https://blog.confiant.com/osx-hydromac-a-new-macos-malware-leaked-from-a-flashcards-app-2af28f1caa9e.

Markstedter, M., 2023. Blue Fox: Arm Assembly Internals and Reverse Engineering. Wiley. ISBN 978-1-119-74530-3. https://books.google.com/books?id=NY6UzQEACAAJ.

Mechtinger, Avigayil, Robinson, Ryan, Fishbein, Nicole, 2022. New SysJoker backdoor targets Windows, Linux, and macOS. https://www.intezer.com/blog/malware-analysis/new-backdoor-sysjoker/.

Myers, Lysa, 2012. More on osx/crisis — advanced spy tool. https://www.intego.com/mac-security-blog/more-on-osxcrisis-advanced-spy-tool/.

Nakagawa, Koh M., 2021. Project Champollion: reverse engineering Rosetta 2. https://ffri.github.io/ProjectChampollion/.

osxreverser, 2020. The finfisher tales, chapter 1: the dropper. https://reverse.put.as/2020/09/26/the-finfisher-tales-chapter-1/.

Owens, Cedric, 2021. macos gatekeeper bypass (2021 edition). https://cedowens.medium.com/macos-gatekeeper-bypass-2021-edition-5256a2955508.

Parallels, 2020. Network modes in parallels desktop for mac. https://kb.parallels.com/4948.

Scarborough, Casey, 2021. macos keylogger. https://github.com/caseyscarborough/keylogger.

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

Unix, 2010. arch(1) [osx man page]. https://www.unix.com/man-page/osx/1/arch/.

Wardle, Patrick, 2017. Osx/proton.b - a brief analysis, at 6 miles up. https://objective-see.com/blog/blog_0x1F.html.

Wardle, Patrick, 2020. Osx.evilquest uncovered. https://objective-see.com/blog/blog_0x59.html.

Wardle, Patrick, 2021a. All your macs are belong to us. https://objective-see.com/blog/blog_0x64.html.

Wardle, Patrick, 2021b. Made in China: Osx.zuru. https://objective-see.com/blog/blog_0x66.html.

Wardle, Patrick, 2021c. Arm'd & dangerous an introduction to analysing arm64 malware targeting macos. In: VB2021 Localhost, p. 3.

Wardle, Patrick, 2022. The Art of Mac Malware: The Guide to Analyzing Malicious Software. No Starch Press.

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