Exploiting Reversing (ER) series: Article 04 (macOS/iOS – part 01)

(a step-by-step vulnerability research series on Win, macOS, hypervisors, browsers, and others)

by Alexandre Borges
release date: FEB/04/2025 | rev: A

00. Quote

"You can't go far in this world by relying on people. People are loyal until it seems opportune not to be."
(Julian Assange played by Benedict Cumberbatch | "The Fifth Estate" movie – 2013)

01. Introduction

Welcome to the fourth article of **Exploiting Reversing (ER) series**, **a step-by-step vulnerability research series on Windows, macOS, hypervisors and browsers**, where we will review concepts, architecture and practical steps related to vulnerability research. My last articles are listed below:

- **ERS_03:** https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/
- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01 :** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/
- **MAS_10:** https://exploitreversing.com/2025/01/15/malware-analysis-series-mas-article-10/
- **MAS_09:** https://exploitreversing.com/2025/01/08/malware-analysis-series-mas-article-09/
- **MAS_08:** https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/
- **MAS_07:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_06:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_05:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_04:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_03:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_02:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_01**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

No doubt, this text is a quite **introductory, step-by-step article** on macOS/iOS because it is a really hard and complex subject, but I need to start it from some point and before drafting new deep articles, it is recommended to review basic concepts, commands, and architecture's details to build a minimal foundation. Obviously, it would be impossible to cover details on each topic, but as now I am focused on vulnerability research of Windows, iOS, and Android operating systems, so certainly I will have enough opportunities and, hopefully, time so this article could work as reasonable review. However, I need to highlight that I left out of this text the recent developments on macOS/iOS and also respective security measure because too many details are always harmful at first approach. Therefore, as expected, this article provides you with a basic introduction to macOS/iOS.

02. Acknowledgments

The year is 2025. Four years ago, I started drafting articles with the sole purpose of helping the hacking and information security community, and as I could already imagine at that time, it would be challenging to find time to continue producing content, and indeed this side effect has been confirmed over time. As I have been using IDA Pro for a long time, I needed a license of my own, and that was when **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)** decided to help me, and since then they have provided continuous and decisive support to write this **Malware Analysis Series (MAS)**, which is focused on malware analysis, and the **Exploiting Reversing series (ERs)**, which is my **current and long-term series** on internals, vulnerability research and, eventually, exploitation in critical topics such as Windows, kernel drivers, macOS, browser and hypervisors.

Time flies, and companies around the world have become more demanding in terms of technical skills, but I still believe that one of the most effective ways to help these professionals is to write articles because such content can offer a solid method to learn details that would be a bit more difficult in live conferences or even other media. I still face serious time constraints to write, but I keep trying to do it because, in some way, I know that these series have been important for people's careers.

Life may be short, but every moment is worth it. Enjoy the journey and keep exploiting it!

03. Lab Infrastructure

This article and next ones I will be using the following lab configuration:

- **Two virtual machines running macOS with XCode installed**.
- **(optional) Virtual machine running Ubuntu 24.04.x LTS:** https://ubuntu.com/download/desktop
- **IDA Pro or IDA Home version (@HexRaysSA):** https://hex-rays.com/ida-pro/
- **010 Editor:** https://www.sweetscape.com/010editor/
- **Malwoverview:** https://github.com/alexandreborges/malwoverview
- **(optional) MacBook Air with M-processor** running macOS Sonoma with XCode installed.

There are a few considerations here:

- In different moments I will be using an **M-machine (MacBook M2/M3/M4)**, which uses arm64 processor and, unfortunately, it is not an optional system in a few moments.
- At the time I am authoring this article, I am using macOS Sonoma.
- Different programs can be installed through the article.
- In a future continuation of this macOS/iOS article, I may be using a jailbroken device.
- Although I will not present here, there are many available articles explaining how to create macOS virtual machines using VMware, Parallels, VirtualBox and Hyper-V.

Of course, if readers do not have IDA Pro, you can follow the article using Ghidra, Binary Ninja or Hopper. At the same way, readers might use macOS running on Intel or Apple Silicon processors., even my advice is always to choose an Apple Silicon device.

04. Resources

In this article we will review a series of topics, but soon readers start their investigation and research on macOS/iOS internals, you will realize that there are a series of traditional approaches and tools that will are useful to accomplish the practical tasks in daily work. In fact, such tools and hints can help you to retrieve basic and valuable information through a more practical way. Personally I think that the **ipsw project** (https://blacktop.github.io/ipsw/ and https://github.com/blacktop/ipsw) offers one of most useful tool to work with macOS and iOS, and probably other researchers also agree about it. We can install it on macOS and Linux, respectively, executing the following command:

- **brew install blacktop/tap/ipsw** (macOS)
- **sudo snap install ipsw** (Linux)

Composing IDA Pro with **ipsw tool** is one of the better ways to retrieve and analyze excellent information, but readers can do some steps manually whether they want to do it. During your research on macOS, you find two important artifacts that provide you with valuable information: **dyld_shared_cache** and **kernelcache**. The **dyld_shared_cached**, which is not actually a cache itself, it is a composition of pre-linked dynamic libraries wrapped up into a single file, which all respective original files have been removed. The **kernelcache** is a single and compressed file composed by the kernel and all respective extensions, and it may become a great source of information during the research process, even though **all symbols have been stripped off** over years. For example, extracting mentioned libraries from the **dyld_shared_cache** is useful for getting libraries to either apply reverse engineering on the frameworks' code or even using them to the linking phase on XCode. If readers are using IDA Pro, a third-party application for extracting **dyld_shared_cache** (as **ipsw, dyld-shared-cache-extractor, jtool2** and other ones) are not necessary for most scenarios, even though such tools are excellent for multiple other situations. Anyway, being able to investigate and even debug are crucial to understanding how iOS and macOS internal mechanisms work.

Only to avoid leaving things as a theoretical exercise, readers might download an IPSW file from one of existing websites (https://ipsw.me), unzip it and extract the dyld_shared_cache. In my simple example, I downloaded iOS 17.5.1 (21F90) for iPhone 15 Max, but you might choose any other one.

If readers do not want to navigate to the website, there is a much easier way to get the same firmware by using **ipsw tool**:

- **ipsw download ipsw --device iPhone16,2 --build 21F90**

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw download ipsw --device iPhone16,2 --build 21F90
   • Getting IPSW              build=21F90 device=iPhone16,2 signed=true version=17.5.1
       7.71 GiB / 7.71 GiB [============================================================| ✅ ] 42.18 MiB/s
     • verifying sha1sum...
   • Created: iPhone16,2_17.5.1_21F90_Restore.ipsw
```

[Figure 01]: Using IPSW to download an iOS firmware

We can retrieve meta-information about the downloaded file, which can be useful as reference, and it is done by executing **ipsw info iPhone16,2_17.5.1_21F90_Restore.ipsw**, as shown below:

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw info iPhone16,2_17.5.1_21F90_Restore.ipsw

[IPSW Info]
===========
Version        = 17.5.1
BuildVersion   = 21F90
OS Type        = Production
FileSystem     = 092-64733-001.dmg
SystemOS       = 092-64752-001.dmg
AppOS          = 090-24896-117.dmg
RestoreRamDisk = [090-24459-117.dmg 090-24511-117.dmg]

Devices
-------

iPhone 15 Pro Max
 > iPhone16,2_D84AP_21F90
   - TimeStamp: 09 May 2024 18:52:05 PDT
   - KernelCache: kernelcache.release.iphone16
   - CPU: A17 Pro (), ID: t8130
   - BootLoaders
       * iBEC.d84.RELEASE.im4p
       * iBoot.d84.RELEASE.im4p
       * iBSS.d84.RELEASE.im4p
       * LLB.d84.RELEASE.im4p
       * sep-firmware.d84.RELEASE.im4p
```

[Figure 02]: Checking information of IPSW file

If readers want to **unpack and extract dyld_shared_cache** or the **kernelcache** (once again, it only is a single and compressed file composed by the kernel and all respective extensions), there are different methods to do it. One of them is using the **ipsw tool**, when we can execute the following command:

- **ipsw extract --dyld iPhone16,2_17.5.1_21F90_Restore.ipsw**, and we get:

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw extract --dyld iPhone16,2_17.5.1_21F90_Restore.ipsw
   • Extracting dyld_shared_cache
      • Mounting DMG 092-64752-001.dmg
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.01
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.02
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.03
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.04
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.05
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.06
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.07
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.08
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.09
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.10
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.11
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.12
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.13
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.14
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.15
      • Created 21F90__iPhone16,2/dyld_shared_cache_arm64e.16
```

[Figure 03]: Using IPSW to extract the dyld_shared_cache

Actually, the result of the extraction is composed of a series of files, which varies depending on the chosen firmware, and such files can be checked below:

```
alexandreborges@alexandremacos artifacts_ipsw % cd 21F90__iPhone16,2
alexandreborges@alexandremacos 21F90__iPhone16,2 % ls
dyld_shared_cache_arm64e                    dyld_shared_cache_arm64e.29
dyld_shared_cache_arm64e.01                 dyld_shared_cache_arm64e.30
dyld_shared_cache_arm64e.02                 dyld_shared_cache_arm64e.31
dyld_shared_cache_arm64e.03                 dyld_shared_cache_arm64e.32
dyld_shared_cache_arm64e.04                 dyld_shared_cache_arm64e.33.dylddata
dyld_shared_cache_arm64e.05                 dyld_shared_cache_arm64e.34.dyldlinkedit
dyld_shared_cache_arm64e.06                 dyld_shared_cache_arm64e.35
dyld_shared_cache_arm64e.07                 dyld_shared_cache_arm64e.36
dyld_shared_cache_arm64e.08                 dyld_shared_cache_arm64e.37
dyld_shared_cache_arm64e.09                 dyld_shared_cache_arm64e.38
dyld_shared_cache_arm64e.10                 dyld_shared_cache_arm64e.39
dyld_shared_cache_arm64e.11                 dyld_shared_cache_arm64e.40
dyld_shared_cache_arm64e.12                 dyld_shared_cache_arm64e.41
dyld_shared_cache_arm64e.13                 dyld_shared_cache_arm64e.42
dyld_shared_cache_arm64e.14                 dyld_shared_cache_arm64e.43
dyld_shared_cache_arm64e.15                 dyld_shared_cache_arm64e.44
dyld_shared_cache_arm64e.16                 dyld_shared_cache_arm64e.45
dyld_shared_cache_arm64e.17                 dyld_shared_cache_arm64e.46
dyld_shared_cache_arm64e.18                 dyld_shared_cache_arm64e.47
dyld_shared_cache_arm64e.19                 dyld_shared_cache_arm64e.48
dyld_shared_cache_arm64e.20                 dyld_shared_cache_arm64e.49
dyld_shared_cache_arm64e.21                 dyld_shared_cache_arm64e.50
dyld_shared_cache_arm64e.22                 dyld_shared_cache_arm64e.51
dyld_shared_cache_arm64e.23                 dyld_shared_cache_arm64e.52
dyld_shared_cache_arm64e.24                 dyld_shared_cache_arm64e.53
dyld_shared_cache_arm64e.25                 dyld_shared_cache_arm64e.54
dyld_shared_cache_arm64e.26                 dyld_shared_cache_arm64e.55.dylddata
dyld_shared_cache_arm64e.27                 dyld_shared_cache_arm64e.56.dyldlinkedit
dyld_shared_cache_arm64e.28                 dyld_shared_cache_arm64e.symbols
alexandreborges@alexandremacos 21F90__iPhone16,2 %
```

[Figure 04]: Extracted dyld_shared_cache files

At the same way, we can extract and uncompress the **kernelcache** file by running:

- **ipsw extract --kernel iPhone16,2_17.5.1_21F90_Restore.ipsw**

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw extract --kernel iPhone16,2_17.5.1_21F90_Restore.ipsw
   • Extracting kernelcache
      • Created 21F90__iPhone16,2/kernelcache.release.iPhone16,2
alexandreborges@alexandremacos artifacts_ipsw %
alexandreborges@alexandremacos artifacts_ipsw % file 21F90__iPhone16,2/kernelcache.release.iPhone16,2
21F90__iPhone16,2/kernelcache.release.iPhone16,2: Mach-O 64-bit arm64e
alexandreborges@alexandremacos artifacts_ipsw %
alexandreborges@alexandremacos artifacts_ipsw % ls -lh 21F90__iPhone16,2/kernelcache.release.iPhone16,2
-rw-r-----  1 alexandreborges  staff    57M Aug  3 11:23 21F90__iPhone16,2/kernelcache.release.iPhone16,2
alexandreborges@alexandremacos artifacts_ipsw %
```

[Figure 05]: Extracted and decompressed kernelcache file

Checking information about the **extracted kernelcache** is done by running:

- **ipsw macho info 21F90__iPhone16,2/kernelcache.release.iPhone16,2 | head -25**

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw macho info 21F90__iPhone16,2/kernelcache.release.iPhone16,2 | head -25
Magic         = 64-bit MachO
Type          = FILESET
CPU           = AARCH64, ARM64e
Commands      = 267 (Size: 19080)
Flags         = None
000: LC_UUID                    D6351B91-EFF9-75A3-CA1D-D8F02C158E13
001: LC_BUILD_VERSION           Platform: Unknown, SDK: 0
002: LC_UNIXTHREAD              Threads: 1
       ARM_THREAD_STATE64 Entry: 0xfffffff00a3dc000
003: LC_DYLD_CHAINED_FIXUPS     offset=0x003898000  size=0x440
004: LC_SEGMENT_64 sz=0x00008000 off=0x00000000-0x00008000 addr=0xfffffff007004000-0xfffffff00700c000 r--/r--   __TEXT
005: LC_SEGMENT_64 sz=0x008f0000 off=0x00008000-0x008f8000 addr=0xfffffff00700c000-0xfffffff0078fc000 r--/r--   __PRELINK_TEXT
        sz=0x008f0000 off=0x00008000-0x008f8000 addr=0xfffffff00700c000-0xfffffff0078fc000       __PRELINK_TEXT.__text
eInstructions|SomeInstructions
006: LC_SEGMENT_64 sz=0x004d4000 off=0x008f8000-0x00dcc000 addr=0xfffffff0078fc000-0xfffffff007dd0000 r--/r--   __DATA_CONST
007: LC_SEGMENT_64 sz=0x0003c000 off=0x00dcc000-0x00e08000 addr=0xfffffff007dd0000-0xfffffff007e0c000 r--/r--   __DATA_SPTM
008: LC_SEGMENT_64 sz=0x025d0000 off=0x00e08000-0x033d8000 addr=0xfffffff007e0c000-0xfffffff00a3dc000 r-x/r-x   __TEXT_EXEC
009: LC_SEGMENT_64 sz=0x00008000 off=0x033d8000-0x033e0000 addr=0xfffffff00a3dc000-0xfffffff00a3e4000 r-x/r-x   __TEXT_BOOT_EXEC
010: LC_SEGMENT_64 sz=0x001ec000 off=0x033e0000-0x035cc000 addr=0xfffffff00a3e4000-0xfffffff00a5d0000 rw-/rw-   __PRELINK_INFO
        sz=0x001ec000 off=0x033e0000-0x035cc000 addr=0xfffffff00a3e4000-0xfffffff00a5d0000       __PRELINK_INFO.__info
011: LC_SEGMENT_64 sz=0x00250000 off=0x035cc000-0x0381c000 addr=0xfffffff00a5d0000-0xfffffff00a820000 rw-/rw-   __DATA
012: LC_SEGMENT_64 sz=0x00080000 off=0x0381c000-0x0389c000 addr=0xfffffff00a820000-0xfffffff00a8a0000 r--/r--   __LINKEDIT
013: LC_FILESET_ENTRY           offset=0x000008000 addr=0xfffffff00700c000 com.apple.kernel
014: LC_FILESET_ENTRY           offset=0x0000c8000 addr=0xfffffff0070cc000 com.apple.AGXFirmwareKextG16PRTBuddy
015: LC_FILESET_ENTRY           offset=0x0000c8d70 addr=0xfffffff0070ccd70 com.apple.AGXFirmwareKextRTBuddy64
016: LC_FILESET_ENTRY           offset=0x0000ccd70 addr=0xfffffff0070d0d70 com.apple.AGXG16P
alexandreborges@alexandremacos artifacts_ipsw %
```

[Figure 06]: kernelcache binary information

We can list and even extract its **respective kernel extensions from the uncompressed kernelcache**:

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw kernel kexts 21F90__iPhone16,2/kerne
lcache.release.iPhone16,2| head -10
   • Kexts                       count=272
0x7fffffffffffff: com.apple.driver.AppleBSDKextStarterTMPFS (1)
0x7fffffffffffff: com.apple.driver.AppleBSDKextStarterVPN (3)
0x7fffffffffffff: com.apple.driver.AppleHIDKeyboardEmbedded (1.2.0a3)
0x7fffffffffffff: com.apple.driver.AppleStorageDrivers (556)
0x7fffffffffffff: com.apple.driver.AppleTopCaseActuatorHIDDriver (7150.1)
0x7fffffffffffff: com.apple.driver.AppleTopCaseDriverV2 (7150.1)
0x7fffffffffffff: com.apple.driver.SCSIDeviceSpecifics (556)
0x7fffffffffffff: com.apple.driver.USBStorageDeviceSpecifics (556)
0x7fffffffffffff: com.apple.driver.usb.AppleUSBHostPlatformProperties (1.2)
0x7fffffffffffff: com.apple.driver.usb.IOUSBHostHIDDeviceSafeBoot (1.2)
alexandreborges@alexandremacos artifacts_ipsw %
alexandreborges@alexandremacos artifacts_ipsw % ipsw kernel extract 21F90__iPhone16,2/ker
nelcache.release.iPhone16,2 --all --output 21F90__iPhone16,2/extracted_kexts
   • Extracting all KEXTs...
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.kernel
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.AGXFirmwareKextG16PRTBuddy
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.AGXFirmwareKextRTBuddy64
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.AGXG16P
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.driver.AOPTouchKext
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.driver.ASIOKit
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.AUC
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.driver.AppleA7IOP-ASCWrap-v6
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.driver.AppleA7IOP
      • Created 21F90__iPhone16,2/extracted_kexts/com.apple.driver.AppleALSColorSensor
```

[Figure 07]: listing and extracting kernel extensions (kexts) from a kernel cache

The **dyld_shared_cache** file can also be parsed:

```
alexandreborges@alexandremacos artifacts_ipsw % ipsw dyld info -l -s 21F90__iPhone16,2/dyld_shared
_cache_arm64e | head -25
Header
======
Magic          = "dyld_v1  arm64e"
Platform       = iOS
OS Version     = 17.5
Max Slide      = 0x20000000 (ASLR entropy: 15-bits, 512MB)
Num Images     = 3431
Num SubCaches  = 56

Shared Region:                  3GB, address: 0x180000000 -> 0x25D0D8000
Local Symbols (nlist array): 155MB, offset:  0x000011690 -> 0x009B4F0D0
Local Symbols (string pool): 508MB, offset:  0x009B4F0D0 -> 0x0297E8FF9
ImagesText Info:             107KB, offset:  0x000000270 -> 0x00001AF50
Patch Info:                  102MB, address: 0x244D51948 -> 0x24B35CBAA
Prebuilt Loader Set:           2MB, address: 0x24B35CBB0 -> 0x00028DC10
Prebuilt Loader Pool:         30MB, address: 0x24B5EA7C0 -> 0x24D4947C0
Dynamic Config:               16kB, address: 0x25D0D4000 -> 0x25D0D8000
ObjC Opts:                     32B, offset:  0x07759E368 -> 0x07759E388
Swift Opts:                    0kB, offset:  0x07626C000 -> 0x07626C000

dyld MachO:    address: 0x1AB8BD000
_dyld_start:   address: 0x1AB900FA0

> Cache UUID: A33C56B8-05B4-3CF5-81D3-8C47670D9996

alexandreborges@alexandremacos artifacts_ipsw %
```

[Figure 08]: parsing the dyld_shared_cache

To extract all dynamic libraries from the **dyld_shared_cache** and symbolicates **ObjC runtime info**, execute the command **ipsw dyld extract dyld_shared_cache_arm64e --all --objc -o extracted_dylds**:

```
alexandreborges@alexandremacos 21F90__iPhone16,2 % ipsw dyld extract dyld_shared_cache_arm64e --all --ob
jc -o
 extracted_dylds
   • Extracting all dylibs from dyld_shared_cache_arm64e
   • Adding ObjC symbols
         0s [--------------------------------------------------------------------------------| 0/3431 ]
      34m20s [--------------------------------------------------------------------------------| 1/3431 ]
      12m51s [--------------------------------------------------------------------------------| 4/3431 ]
      15m0s [--------------------------------------------------------------------------------| 4/3431 ]
      11m25s [--------------------------------------------------------------------------------| 6/3431 ]
      11m0s [--------------------------------------------------------------------------------| 7/3431 ]
   • Adding ObjC symbols
      9m30s [--------------------------------------------------------------------------------| 9/3431 ]
      10m29s [--------------------------------------------------------------------------------| 9/3431 ]
   • Adding ObjC symbols
      9m19s [--------------------------------------------------------------------------------| 11/3431 ]
      9m15s [--------------------------------------------------------------------------------| 12/3431 ]
   • Adding ObjC symbols
      9m6s [--------------------------------------------------------------------------------| 15/3431 ]
      14m56s [--------------------------------------------------------------------------------| 16/3431 ]
      16m32s [--------------------------------------------------------------------------------| 16/3431 ]
      15m15s [--------------------------------------------------------------------------------| 19/3431 ]
      17m57s [--------------------------------------------------------------------------------| 19/3431 ]
```

[Figure 09]: extracting all dynamic libraries from dyld_shared_cache

We can examine the dyld_shared_cache content such as classes, protocols and selectors.

To dump classes and protocols, run **ipsw dyld objc --class dyld_shared_cache_arm64e** and **ipsw dyld objc --proto dyld_shared_cache_arm64e** commands:

```
alexandreborges@alexandremacos 21F90__iPhone16,2 % ipsw dyld objc --class dyld_shared_cache_
arm64e | head -10
0x1e942f118: DBManager  CallHistory
0x23c1af048: IMFullScreenEffect IMSharedUI
0x1ea898760: GKCollectionGridLayout     GameCenterUI
0x23c653eb0: STUIStatusBarAccessibility__SpringBoardFramework__SystemStatusUI   SpringBoard
0x1e9bbc430: _TtC12SiriOntology28UsoEntity_common_Measurement   SiriOntology
0x1ea6ee0d0: SBSUIWallpaperProgressHUD  SpringBoardUIServices
0x1ea069a90: UITextMagnifierCommonRenderer      UIKitCore
0x1eb1967b8: NCClickInteractionPortalView       UserNotificationsUIKit
0x1eaaa5240: _TtC15RemindersUICore32TTRIMenuButtonAttachmentProvider    RemindersUICore
0x23c0a0f40: SXVideoPlayerViewControllerResponse        Silex
alexandreborges@alexandremacos 21F90__iPhone16,2 %
alexandreborges@alexandremacos 21F90__iPhone16,2 % ipsw dyld objc --proto dyld_shared_cache_
arm64e | head -10
0x1e8f26ac8: AKUserInterfaceItem        AnnotationKit
0x1e8dc3c28: ICProgressViewControllerDelegate   NotesUI
0x1e8f08e88: ICQUIRemoteUIPresenterDelegate     iCloudQuotaUI
    0x1e8d7c1c8: CADBackupRestoreInterface      EventKit
    0x1e8d7c1c8: CADBackupRestoreInterface      CalendarDaemon
0x1e8db09c8: ISEffect   IconServices
    0x1e8ee9ec8: TSKAccessControllerDelegate    TSReading
    0x1e8ee9ec8: TSKAccessControllerDelegate    KeynoteQuicklook
    0x1e8ee9ec8: TSKAccessControllerDelegate    TSApplication
    0x1e8ee9ec8: TSKAccessControllerDelegate    PagesQuicklook
alexandreborges@alexandremacos 21F90__iPhone16,2 %
```

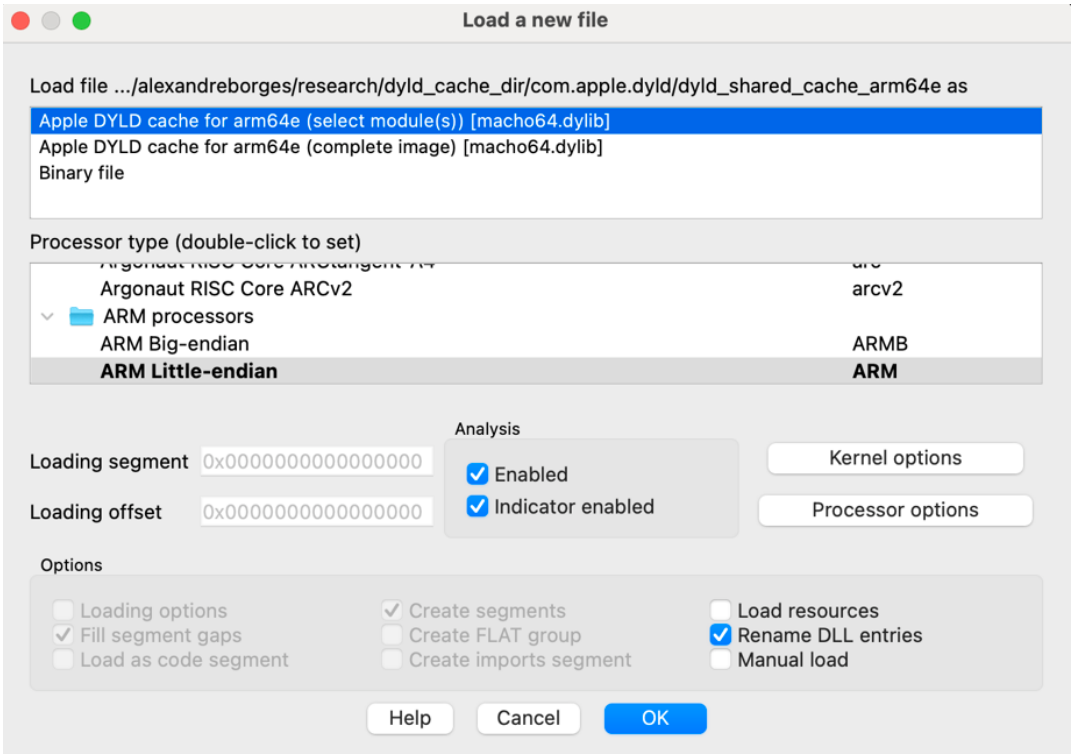
[Figure 10]: Dumping classes and protocols from dyld_shared_cache

You always have the choice of proceeding manually, and doing it once can be interesting, as shown below.

- **mv iPhone16,2_17.5.1_21F90_Restore.ipsw iPhone16,2_17.5.1_21F90_Restore.zip**
- **mkdir unpacked**
- **mv iPhone16,2_17.5.1_21F90_Restore.zip unpacked**
- **unzip iPhone16,2_17.5.1_21F90_Restore.zip**
- **ls -lh**
- **hdiutil attach 092-64752-001.dmg (note: the second largest one)**
- **mkdir shared_cache_dir**
- **cp -r /Volumes/DawnF21F90.D84SystemCryptex/System/Library/Caches/com.apple.dyld shared_cache_dir/**
- **cd shared_cache_dir/**

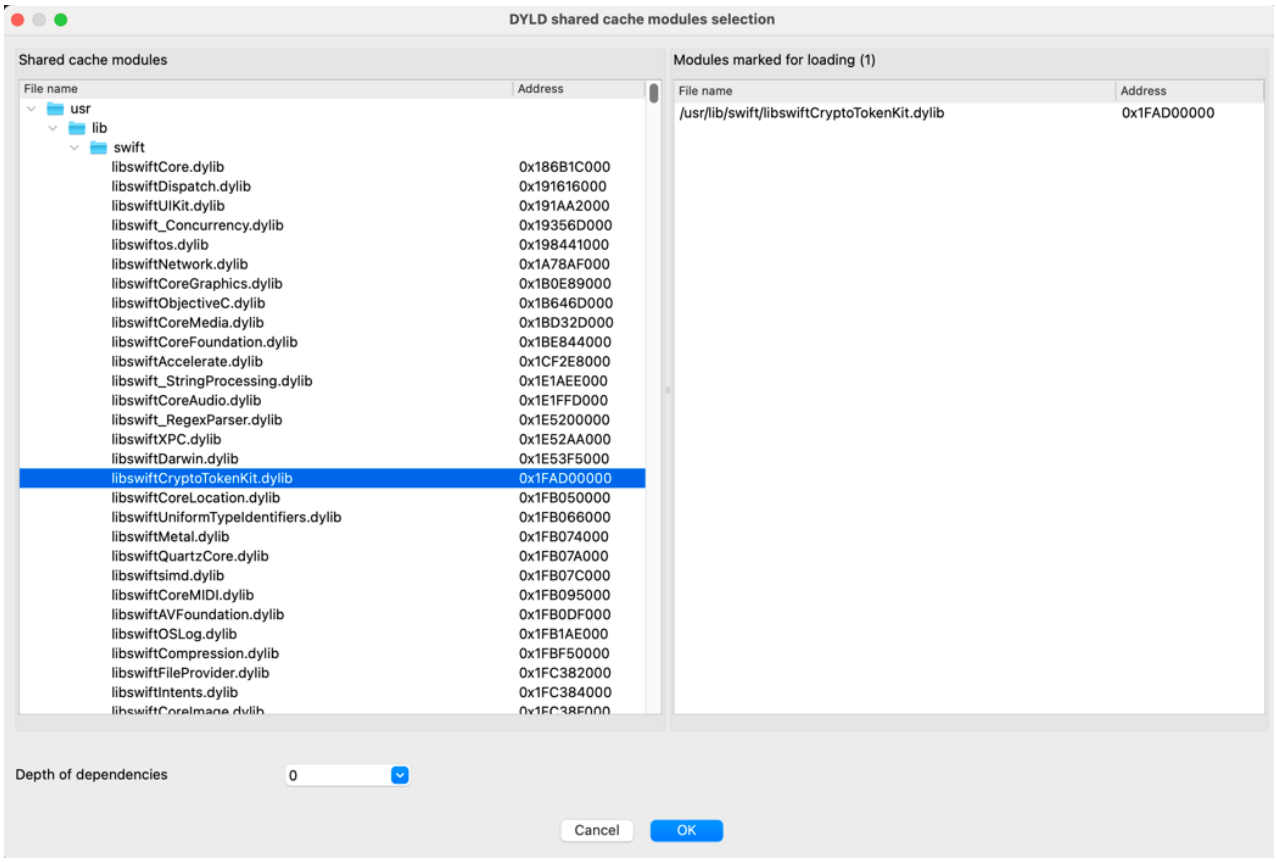
I want to highlight that, in this example, I am using the second largest dmg file, but depending on the image, **dyld_shared_cache** is present in the largest file. I copied the entire folder to my home folder because I will need to manipulate these files, and it will result in a few additional files too. Of course, readers may take the best approach for you.

From this point everything is quite simple, and we can open the first file (**dyld_shared_cache_arm64e**) in the IDA Pro, which will present the following dialog:



[Figure 11]: IDA Pro: opening a dyld_shared_cache

There are options to either choose a specific module (recommended) or pick up the complete image. Choosing the first one, readers will see the following:



[Figure 12]: IDA Pro: selecting a specific module from the dyld_shared_cache

From this point onward, it would be a usual task of reverse engineering, which might be hard sometimes. By the way, if you are not using **IDA** or **ipsw tool**, so we could still extract all modules from the **dyld_shared_cache** manually using an open-source project, and one of possible approaches to do it would be the following one:

- **hdiutil detach /Volumes/DawnF21F90.D84SystemCryptex** (from our previous mounting)
- **brew install keith/formulae/dyld-shared-cache-extractor**
- **dyld-shared-cache-extractor dyld_shared_cache_arm64e /tmp/extracted_dyld_shared_cache**

```
alexandreborges@alexandremacos extracted_dyld_shared_cache % pwd
/tmp/extracted_dyld_shared_cache
alexandreborges@alexandremacos extracted_dyld_shared_cache % tree -d . | head -15
.
|-- System
|   `-- Library
|       |-- AccessibilityBundles
|       |   |-- ARKit.axbundle
|       |   |-- ARTraceModule.axbundle
|       |   |-- ASMessagesProvider.axbundle
|       |   |-- AVFoundation.axbundle
|       |   |-- AVKit.axbundle
|       |   |-- AXActionSheetUIServer.axuiservice
|       |   |-- AXSpeechImplementation.bundle
|       |   |-- AccessibilitySettings.axbundle
|       |   |-- AccessibilitySettingsLoader.bundle
|       |   |-- AccountsUI.axbundle
|       |   |-- AcousticId-Assistant.axbundle
alexandreborges@alexandremacos extracted_dyld_shared_cache %
```

[Figure 13]: dyld_shared_cache extracted using an open-source project

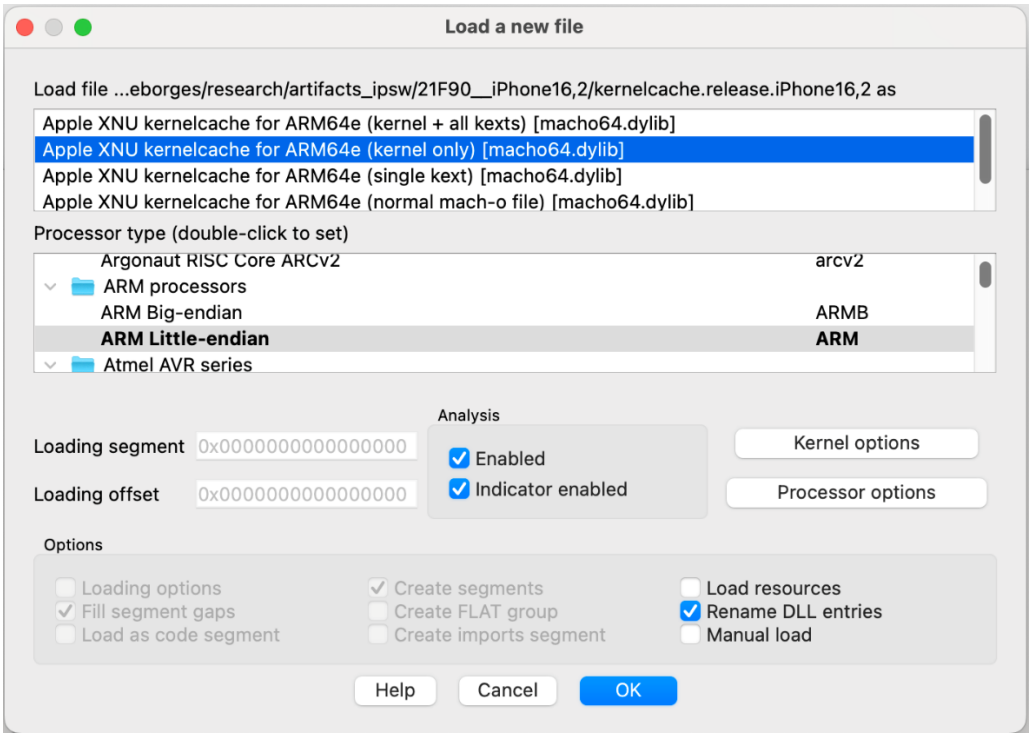
We can see that the extraction process has been worked. If you receive a message *like "xcrun: error: SDK "iphoneos" cannot be located"* , launch the XCode and go to **XCode menu | Settings… | Locations** and, at **Command Line Tools**, select (or re-select) the appropriate XCode version (in my case is Xcode 15.4 (15F31d).

Using a similar procedure, we can open and analyze the extracted kernelcache using IDA Pro, which is clearly better because even whether it is compressed, the decompression is automatically perform and there is the advantage of using Lumina:

When we open an extracted kernelcache with IDA Pro, the following options will be presented, as shown on the next page:

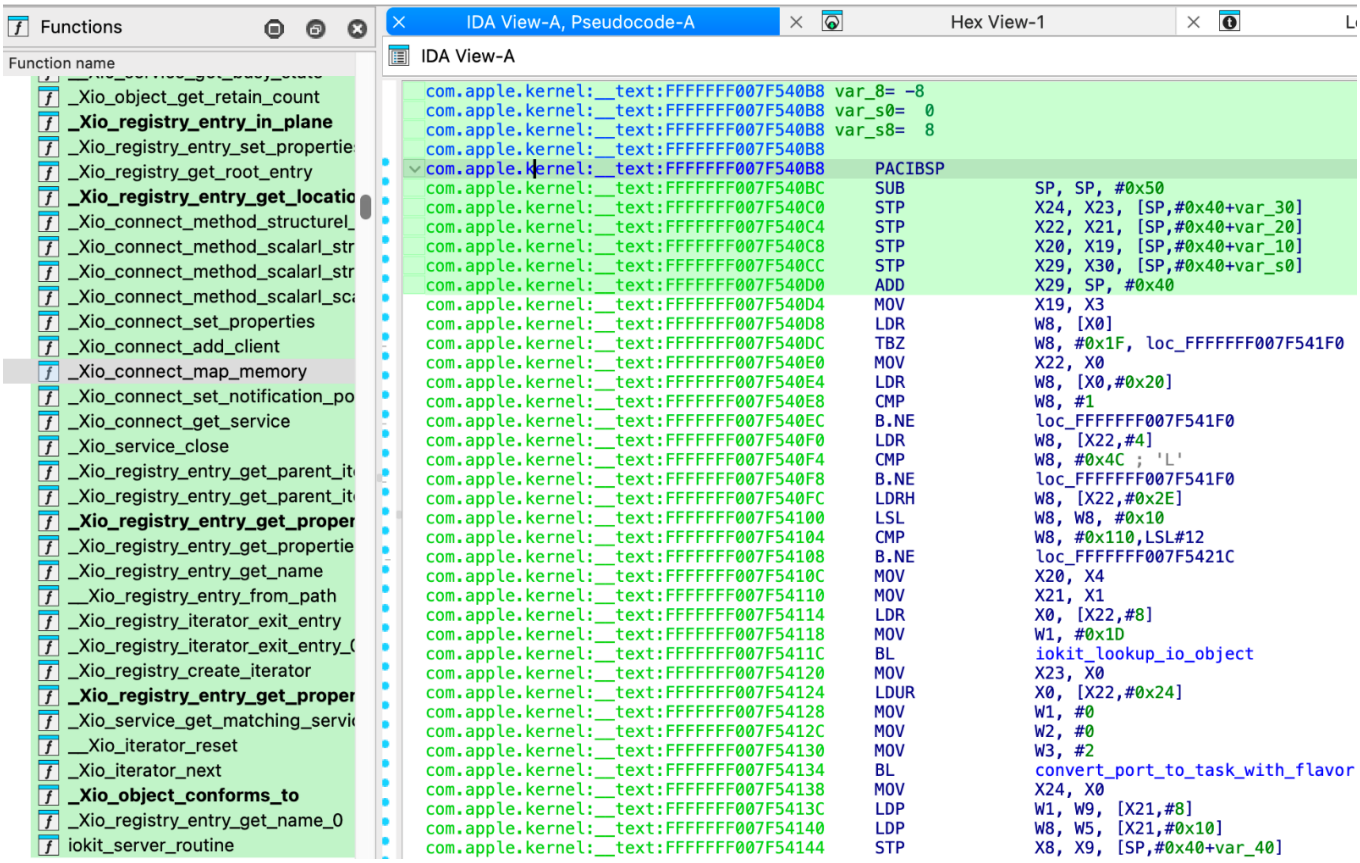
- Apple XNU kernelcache (kernel + all kexts)
- Apple XNU kernelcache (kernel only)
- Apple XNU kernelcache (single kext)
- Apple XNU kernelcache (normal mach-o file)
- Binary File

The choice depends on the context involved, but in many opportunities, we will be interested in analyzing information coming from the kernel itself. Therefore, we are going to use this option here too:



[Figure 14]: Opening kernelcache in IDA Pro

Now we can apply Lumina by navigating to the menu **Lumina | Pull all (F12)**:



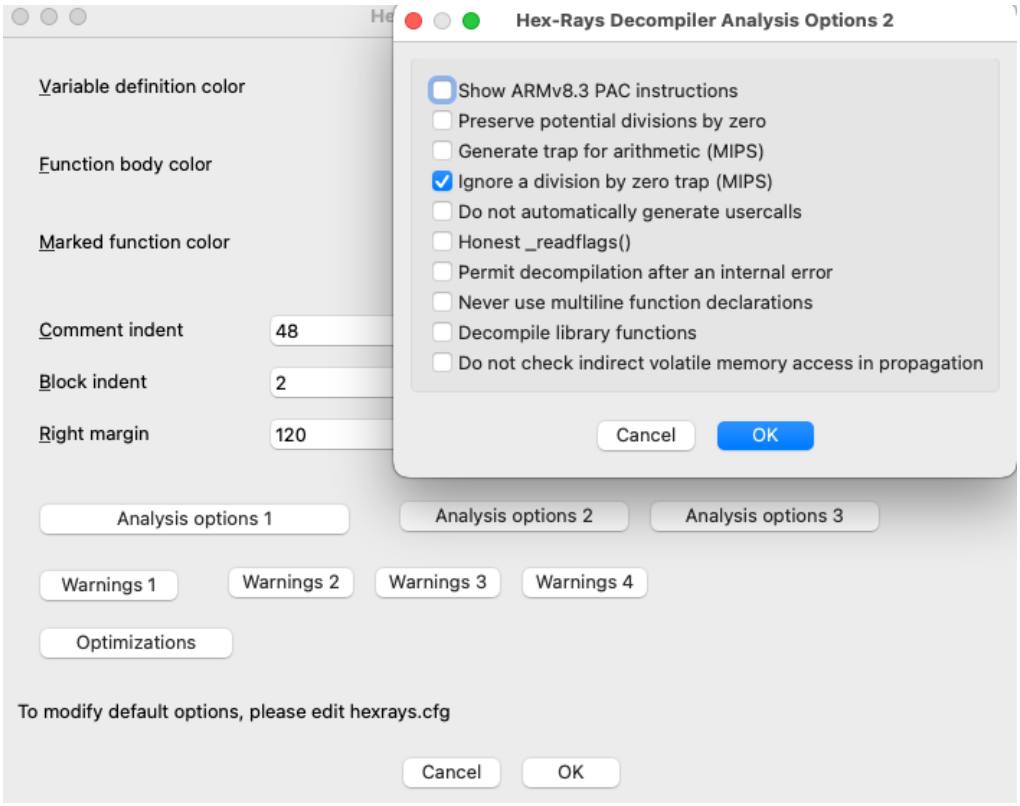
[Figure 15]: Using Lumina over kernelcache

As we can notice, there are a lot of functions commented on, and data types applied thanks to Lumina. The decompilation process is the same as go to **Edit | Produce File | Create a C file**. The task will take some time, and after it has been finished, go to menu **View | Open sub views | Generate pseudocode** (or **F5**), and put it synchronized and side-by-side with the IDA View-A.

Lumina offers fantastic support for recognizing functions that it would be time consuming to discover manually. Although we have not mentioned it, beyond being able to analyze statically we also have the opportunity to dynamically analyze the dyld_shared_cache through the IDA Pro debugger. It offers us a chance to get an insight into libraries that would be available only such a cache and debug the relationship between a given framework and a library from the dyld_shared_cache used by this framework. We will return to these topics, but it is important to underscore that macOS and mainly iOS are extensive and mainly many concepts are necessary to get a minimum understanding of them.

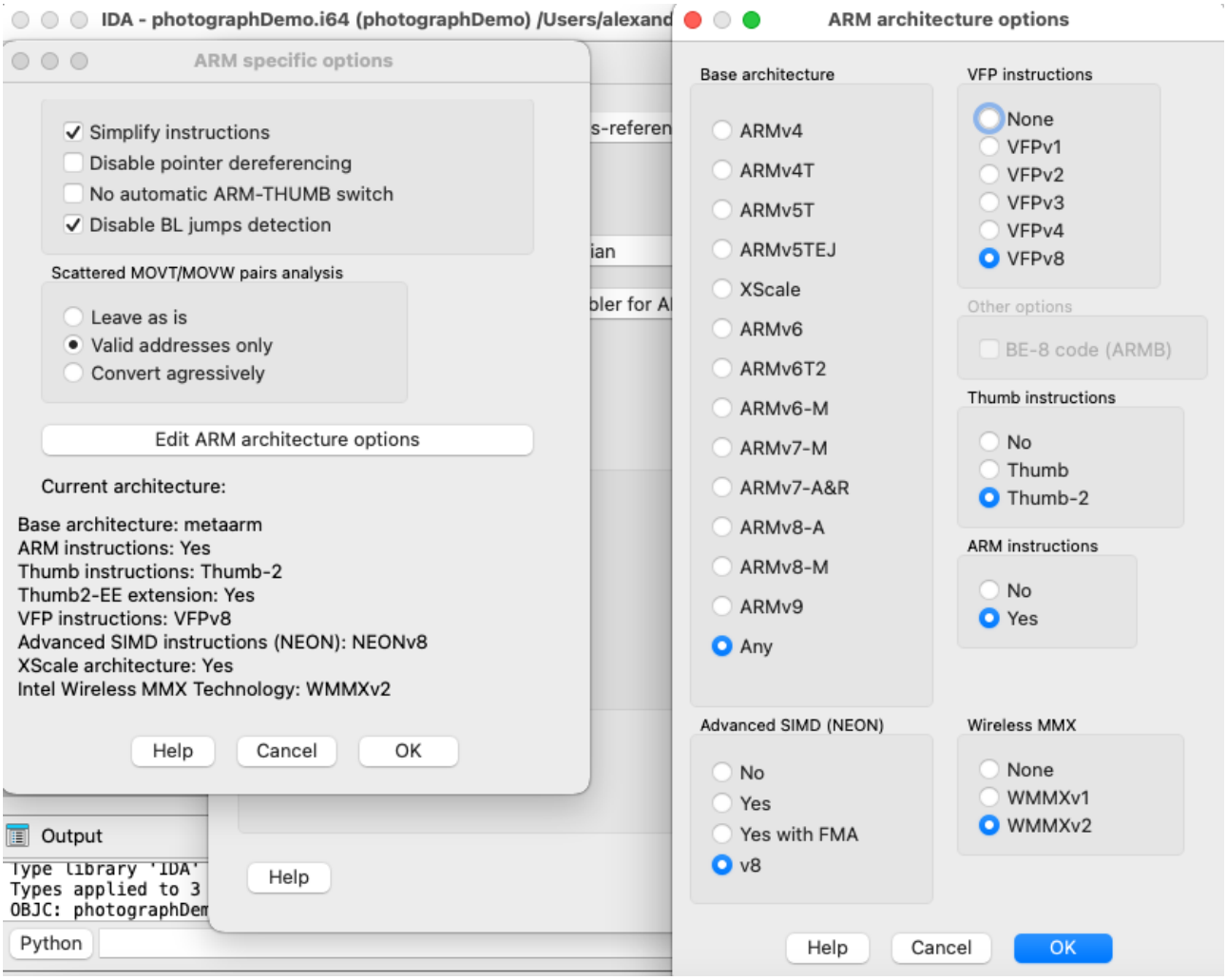
Only two small suggestions follow:

- While decompiling a Mach-O binary, PAC (Pointer Authentication Code) instructions are not shown and, if you need them, go to **Edit | Plugins | Hex-Rays Decompiler | Options. | Analysis options 2** and enable "**Show ARMv8.3 PAC instructions**" option.



[Figure 16]: How to show PAC instruction on decompiled code

- Although IDA automatically choose the correct architecture, you might change the ARM architecture by navigating through the menu **Options | General…| Analysis | Processor specific analysis option | Edit ARM architecture options**



[Figure 17]: Adjusting ARM architecture

Although I have enough space here, readers might try other **ipsw** commands:

- **ipsw device-list:** it presents a list of all iOS devices and mainly their respective architectures.
- **ipsw download macos:** it downloads macOS installers.
- **ipsw mount <sys|app|fs> iPhone16,2_17.5.1_21F90_Restore.ipsw:** this command mount portions (system, application, or filesystem of the given IPSW file.
- **ipsw ctfdump <kernel_development_kit_file>:** it retrieves Compact ANSI-C Type Format (CTF) from a KDK.
- **ipsw download dev:** it allows to download XCode, KDK and other ones from Apple Developers website, but it requires account login.
- **ipsw class-dump [<DSC> <DYLIB>|<MACHO>] [flags]:** ObjC class-dump a dylib from a Mach-O or a DSC (dynamic shared cache) file.
- **ipsw swift-dump [<DSC> <DYLIB>|<MACHO>] [flags]:** Swift class-dump a dylib from a Mach-O or a DSC (dynamic shared cache)
- **ipsw update:** it updates the own ipsw tool.

It is time to move on, learn how to set up the environment and review concepts.

05. Introduction to XNU kernel debugging using LLDB

While researching macOS, readers may need to debug the macOS kernel and respective extensions. There are a few techniques to accomplish this task, and the choice depends on the infrastructure that macOS is being run such as physical or virtual machines. Over time, and not surprisingly, such methods have slightly changed due to the macOS version and also because Apple has taken different measures against security issues.

As in this article it is using macOS running on VMware virtual machines and also a physical MacBook M2, I am going to start showing how to use the built-in GDB debugger capabilities from VMware, which is one useful and accessible technique to use, and I will try to refrain from presenting unnecessary details. Additionally, I will quickly show another technique using **KDP (Kernel Debugging Protocol)**, which works well, but I prefer using it with physical hosts rather than virtual machines.

Therefore, to start to use the GDB built-in server from VMware (in my case I am using VMware Workstation), the first step is to edit the respective **.vmx file** (virtual machine configuration file) and add a few parameters. The following parameters are available:

- **debugStub.listen.guest64:** enables local virtual machine debugging.
- **debugStub.listen.guest64.remote:** enables remote virtual machine debugging.
- **monitor.debugOnStartGuest64:** enables debugging since the virtual machine start-up.
- **debugStub.hideBreakpoints:** tries to hide hardware breakpoints from the virtual machine
- **debugStub.port.guest64:** establishes a listening port for the GDB stub (the default is 8864).

All these parameters assume a 64-bit virtual machine. The first two parameters are recommended, and should be configured with the following values:

- **debugStub.listen.guest64** = "TRUE"
- **debugStub.listen.guest64.remote** = "TRUE"

The remaining parameters depend on the context. For example, we can want to debug the virtual machine since the power-on, and in this case, we should enable **monitor.debugOnStartGuest**. At the same way, we can want to hide breakpoints (it might cause issues on recent macOS versions) or even change the debugging port, and we would change the last two parameters.

- **monitor.debugOnStartGuest64** = "FALSE"
- **debugStub.hideBreakpoints** = "FALSE"
- **debugStub.port.guest64** = "8864"

For now, I recommend using the parameters as shown above.

Although it would not really be necessary, you have the option to follow the virtual machine log (**vmware.log**), which is located in the same directory as all other virtual-machine files. There are two programs (there are many others) that you could use:

- **Tail for Windows:** https://sourceforge.net/projects/wintail/
- **BareTail:** https://sourceforge.net/p/tailw/wiki/Home/

The log will be the following aspect:

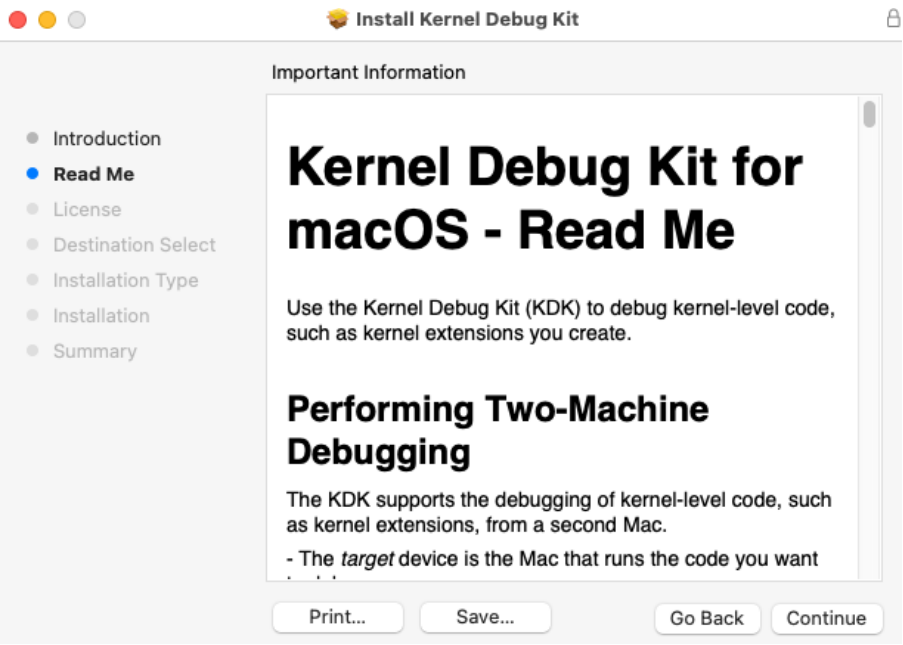
```
2024-08-23T03:34:43.359Z In(05) vcpu-0 Vix: [mainDispatch.c:4212]: VMAutomation_ReportPowerOpFinished: statevar=0, newAppState=1872,
2024-08-23T03:34:43.359Z In(05) vcpu-0 Vix: [mainDispatch.c:4129]: VMAutomationReportPowerStateChange: Reporting power state change
2024-08-23T03:34:43.359Z In(05) vcpu-0 Vix: [mainDispatch.c:4129]: VMAutomationReportPowerStateChange: Reporting power state change
2024-08-23T03:34:43.359Z In(05) vcpu-0 Transitioned vmx/execState/val to poweredOn
2024-08-23T03:34:43.359Z In(05) vcpu-0 Tools: Adding Tools inactivity timer.
2024-08-23T03:34:43.359Z In(05) vcpu-0 Intel VT: FlexPriority disabled, VPID enabled.
2024-08-23T03:34:43.362Z In(05) vmx USB: New set of 9 USB devices.
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:DisplayLink\ Dell\ Universal\ Dock\ D6000 vid:17e9 pid:6006 path:1/3/2/0
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Chicony\ HD\ Webcam vid:04f2 pid:b68b path:1/1/12 speed:high family:vide
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Intel(R)\ Wireless\ Bluetooth(R) vid:8087 pid:0026 path:1/1/13 speed:ful
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Realtek\ USB3.0-CRW vid:0bda pid:0316 path:1/1/17 speed:super family:sto
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Seagate\ RSS\ Expansion\ Desk vid:0bc2 pid:331a path:1/1/20 speed:super
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Seagate\ RSS\ Expansion\ HDD vid:0bc2 pid:2038 path:1/1/21 speed:super f
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Bizlink\ D6000\ Controller vid:06c4 pid:c413 path:1/1/0/3 speed:full fam
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Logitech\ Logi\ USB\ Camera\ (C270\ HD\ WebCam) vid:046d pid:0825 path:1
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Virtual\ Bluetooth\ Adapter vid:0e0f pid:0008 speed:full family:wireless
2024-08-23T03:34:43.363Z No(00) vmx ConfigDB: Setting sata0:1.fileName = <not printed>
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: event, up:1, adapter:0
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: 'ethernet0' state from 4 to 6.
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: event, up:1, adapter:0
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: 'ethernet1' state from 4 to 6.
2024-08-23T03:34:43.383Z In(05) vcpu-0 Vix: [vmxCommands.c:4556]: VMAutomation_Pause: pause = TRUE
2024-08-23T03:34:43.384Z In(05) vcpu-0 Vix: [mainDispatch.c:6914]: VMAutomation: Ignoring VMAutomation_SendMsgToOneClient because co
2024-08-23T03:34:43.384Z Wa(03) vcpu-0 Waiting for Guest 64-bit debugger to connect!
```

[Figure 18]: vmware.log

On the log presented, I had configured the guest since its start, but as I mentioned, it is only an example of examining log actions on VMware.

The next step is to download the Kernel Debug Kit (KDK) on both machines (host and target), and the recommendation is to download the KDK matching with running macOS version, which can be checked by running **sw_vers** command. As an example, at the time I am drafting this article, my both experimental system run Sonoma 14.6.1 (23G93) and I downloaded exactly this version of KDK from https://developer.apple.com/download/more/ website, or use the **ipsw** command that was quickly presented in the previous section. The KDK contains development and, eventually, debug version of the macOS kernel (unfortunately, it has not happened for some time), and inside of this package you will find symbolic information, LLDB macros and other resources.

To install the KDK we could use the command line, but it is easier to double-click the **.dmg file** and you will see the following window:



[Figure 19]: Installing KDK

The KDK is installed on **/Library/Developer/KDKs** directory and, for the version I am using, it has been created a **KDK_14.6.1_23G93.kdk** sub-directory there, which contains a series of resources. In Kernels directory, we find the following files:

```
alexandreborges@Mac Library % pwd
/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library
alexandreborges@Mac Library %
alexandreborges@Mac Library % tree -d . -L 1
.
├── CoreServices
├── Extensions
├── KernelCollections
├── KernelSupport
└── Kernels

6 directories
alexandreborges@Mac Library % ls Kernels
kernel                          kernel.development.vmapple      kernel.kasan.vmapple
kernel.dSYM                     kernel.development.vmapple.dSYM kernel.kasan.vmapple.dSYM
kernel.development              kernel.kasan                    kernel.release.t6000
kernel.development.dSYM         kernel.kasan.dSYM               kernel.release.t6000.dSYM
kernel.development.t6000        kernel.kasan.t6000              kernel.release.t6020
kernel.development.t6000.dSYM   kernel.kasan.t6000.dSYM         kernel.release.t6020.dSYM
kernel.development.t6020        kernel.kasan.t6020              kernel.release.t6030
kernel.development.t6020.dSYM   kernel.kasan.t6020.dSYM         kernel.release.t6030.dSYM
kernel.development.t6030        kernel.kasan.t6030              kernel.release.t6031
kernel.development.t6030.dSYM   kernel.kasan.t6030.dSYM         kernel.release.t6031.dSYM
kernel.development.t6031        kernel.kasan.t6031              kernel.release.t8103
kernel.development.t6031.dSYM   kernel.kasan.t6031.dSYM         kernel.release.t8103.dSYM
kernel.development.t8103        kernel.kasan.t8103              kernel.release.t8112
kernel.development.t8103.dSYM   kernel.kasan.t8103.dSYM         kernel.release.t8112.dSYM
kernel.development.t8112        kernel.kasan.t8112              kernel.release.t8122
kernel.development.t8112.dSYM   kernel.kasan.t8112.dSYM         kernel.release.t8122.dSYM
kernel.development.t8122        kernel.kasan.t8122              kernel.release.vmapple
kernel.development.t8122.dSYM   kernel.kasan.t8122.dSYM         kernel.release.vmapple.dSYM
```

[Figure 20]: Part of the KDK content

It is interesting to notice that there are the release and development versions of the kernel, but there is not any debug version, as I mentioned previously. The following procedure can be adapted accordingly to the environment, goals and even techniques used to debug kernel, and we will adapt it when necessary. It is assumed that you have already made changes in **vmx file** of the target virtual machine as I showed previously. It is important to say that disable SIP is not required to many cases (mainly this one), but I will do it to prepare the environment to next procedures. Only to confirm, my current environment at the time I am drafting this article (Sequoia has been released and I am using in my main machines) is:

- **host:** macOS Sonoma 14.6.1
- **target:** macOS Sonoma 14.6.1
- **virtualization product:** VMware Workstation 17.5.2 build-23775571
- **Xcode is installed on both virtual machines**

Therefore, to debug the kernel of the target macOS execute the following:

1. (host and target) Discover the exact macOS version: **sw_vers | grep BuildVersion**
2. (host and target) Confirm that you downloaded and installed the KDK.
3. (target) Run: **csrutil status** to check if SIP is enabled.

4. (target) If SIP is enabled (default), so reboot the macOS to the **Recovery System mode** (**Command+R**). From the **Utilities menu**, go to **Terminal**. As we are using a VMware virtual machine, we have to use the **macOS iso file** (for example, Sonoma) to boot in Recovery System mode. If we were using real Apple hardware, it would be enough to press **Command+R** during the initial seconds of the system boot.
5. (target) If the machine has **Apple T2 Security Chip**, you will have to change **Secure Boot policy** to "**Medium Security".**
6. (target) Run **csrutil disable** to disable SIP (disable SIP – System Integrity Protocol)
7. (target) Reboot the system: **reboot**
8. (target) Confirm that SIP is disabled by running **csrutil status.**
9. (host) It is time to connect to the **GDB server from VMware Workstation** from the host macOS. In my case, it is running on 192.168.0.96:8864, and we can use lldb or gdb (**brew install gdb**):
    a. **lldb**
    b. (lldb) **target create /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel**
    c. (lldb) **settings set target.load-script-from-symbol-file true**
    d. (lldb) **gdb-remote 192.168.0.96:8864**

```
alexandreborges@Mac ~ % lldb
(lldb) target create /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel
warning: 'kernel' contains a debug script. To run this script in this debug session:

    command script import "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.dSYM/Conten
ts/Resources/Python/kernel.py"

To run all discovered debug scripts in this session:

    settings set target.load-script-from-symbol-file true

Current executable set to '/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel' (x86_64).
(lldb) settings set target.load-script-from-symbol-file true
Loading kernel debugging from /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.dSYM/Con
tents/Resources/Python/kernel.py
LLDB version lldb-1500.0.404.7
Apple Swift version 5.10 (swiftlang-5.10.0.13 clang-1500.3.9.4)
settings set target.process.python-os-plugin-path "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Ke
rnels/kernel.dSYM/Contents/Resources/Python/lldbmacros/core/operating_system.py"
settings set target.trap-handler-names hndl_allintrs hndl_alltraps trap_from_kernel hndl_double_fault hndl_machin
e_check _fleh_prefabt _ExceptionVectorsBase _ExceptionVectorsTable _fleh_undef _fleh_dataabt _fleh_irq _fleh_deci
rq _fleh_fiq_generic _fleh_dec
command script import "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.dSYM/Contents/R
esources/Python/lldbmacros/xnu.py"
xnu debug macros loaded successfully. Run showlldbtypesummaries to enable type summaries.
settings set target.process.optimization-warnings false
settings set target.process.experimental.os-plugin-reports-all-threads false
settings set target.process.run-all-threads true


(lldb) gdb-remote 192.168.0.96:8864
Target arch: x86_64
Connected to live debugserver or arm core. Will associate on-core threads to registers reported by server.
Process 1 stopped
* thread #1, stop reason = signal SIGTRAP
    frame #0: 0xffffff800dd01106
->  0xffffff800dd01106: rep    stosb    %al, %es:(%rdi)
    0xffffff800dd01108: retq
    0xffffff800dd01109: nop
    0xffffff800dd0110a: nop
Target 0: (kernel) stopped.
```

[Figure 21]: macOS kernel debugging

A concise explanation on the commands executed:

- **target create <KDP kernels directory>:** indicates the location of the kernel symbols.
- **settings set target.load-script-from-symbol-file true:** it triggers the loading of scripts located inside symbol directories (named dSYM), which provide us with a substantial number of valuable scripts. You could leave this command in **.lldbinit** file, if you wanted to.
- **gdb-remote <IP>:[port]:** it connects to the gdb servers exposed by VMware.

The exact output depends on the point that the system is at the time of the debugging session. In the previous image, the system was booting (it has been resumed by running "**c**" command). After the logon, the connection shows the following image:

```
Process 1 stopped
* thread #1, stop reason = signal SIGTRAP
    frame #0: 0xffffff800e17d847 kernel`machine_idle at pmCPU.c:179:3 [opt]
Target 0: (kernel) stopped.
(lldb) image list
[  0] B8A95C25-3789-30A2-AE3C-7A48E0B91426 0xffffff800dee4000 /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System
/Library/Kernels/kernel
      /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.dSYM/Contents/Resources/DWARF/ke
rnel
[  1] 250D1260-447B-3BF3-96B2-C784699ED806 0xffffff8010368000 /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System
/Library/Extensions/IOPCIFamily.kext/Contents/MacOS/IOPCIFamily
      /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Extensions/IOPCIFamily.kext.dSYM/Contents/Resou
rces/DWARF/IOPCIFamily
[  2] 1EEEAAAA-95C2-3606-8488-591BE79DB5B4 0xffffff801049e000 /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System
/Library/Extensions/IOStorageFamily.kext/Contents/MacOS/IOStorageFamily
      /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Extensions/IOStorageFamily.kext.dSYM/Contents/R
esources/DWARF/IOStorageFamily
[  3] 763392EF-4CBB-3EE6-88E4-B948D44564BE 0xffffff80103b0000 /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System
/Library/Extensions/IOSCSIArchitectureModelFamily.kext/Contents/MacOS/IOSCSIArchitectureModelFamily
      /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Extensions/IOSCSIArchitectureModelFamily.kext.d
SYM/Contents/Resources/DWARF/IOSCSIArchitectureModelFamily
[  4] B0E2C177-CD2E-3B83-814A-D2C6FE27A789 0xffffff80103d8000 /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System
/Library/Extensions/IOSCSIArchitectureModelFamily.kext/Contents/PlugIns/IOSCSIBlockCommandsDevice.kext/Contents/M
acOS/IOSCSIBlockCommandsDevice
```

[Figure 22]: macOS kernel debugging (second part)

About available commands, it is more productive to learn to debug inside a given context than separated commands, and we will have chance to review some ones. Furthermore, I mentioned that readers must install **XCode,** and it would be advisable to install a few Python packages for a better kernel debugging experience, so run the following steps **on host** and restart its debugging session if it is needed:

- (host) **xcode-select –install**
- (host) **xcrun python3 -m pip install --user --ignore-installed macholib**
- (host) **xcrun python3 -m pip install --user --ignore-installed future**

An important detail is that, as we are using the release-version of the kernel then there is no symbol available for us during the debugging session. Officially, Apple may release different versions of kernels, which are also shown in Figure 20:

- **kernel:** it is the release kernel, which is the same kernel running on our macOS systems.
- **kernel.development:** it is slightly changed when compared to kernel (assertions and error checking included)
- **kernel.kasan:** it is a modified kernel with enabled address sanitizers for detect memory issues.

If you want to use one of these versions, you will have to make a few changes to the system, and the first step is to disable SIP, as we already have done previously. Which one should we pick up? The development version is better. The step-by-step to boot an alternative kernel release is shown below, and it basically

repeats the described by the “Kernel Debug Kit for macOS – Read Me” document from Apple, which is present inside the KDK, even though I have made a few comments:

- disable SIP (**csrutil disable**) and Authenticated Root protection (**csrutil authenticated-root disable**). As explained previously, you will have to boot in Recovery Mode to perform both tasks.
- identify the system volume disk device name, which is mounted at “/”: **mount | grep sealed**. In my case, it is /dev/disk1s4
- create a temporary mount directory: **mkdir /Users/ab/livemount** (as it is usually done).
- mount the system volume onto the temporary directory: **sudo mount -o nobrowse -t apfs /dev/disk1s4 /Users/ab/livemount**
- copy the entire System directory from KDK into the System directory of the mounted disk: **sudo ditto /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System /Users/ab/livemount/System**
- rebuild the kernel collections and bless (authorize) all changes (curiously, it searches for kernel.debug and kernel.research, but we do not have them):
  - **sudo kmutil install --volume-root /Users/ab/livemount --update-all**
  - **sudo bless --mount /Users/ab/livemount --bootefi --create-snapshot**
- By the way, readers could have used **kextcache -invalidate /** command to invalidate cache (kextcache is actually deprecated), but now **kmutil** takes care everything. I suggest readers check the kmutil man page for further details.
- change the device’s boot-args to indicate the new kernel version and, in this case, I will use development version:
  - **sudo nvram boot-args=“debug=0x44 kdp_match_name=en0 wdt=-1 kcsuffix=development”**
- Reboot the system: **sudo reboot**
- Check the running macOS version and the variant:
  - **sw_vers**
  - **sysctl kern.osbuildconfig**

```
ab@macOS2 Kernels % sw_vers
ProductName:            macOS
ProductVersion:         14.6.1
BuildVersion:           23G93
ab@macOS2 Kernels %
ab@macOS2 Kernels % sysctl kern.osbuildconfig
kern.osbuildconfig: development
ab@macOS2 Kernels %
```

[Figure 22]: macOS development kernel variant

Some considerations about the procedure above:

- The **csrutil authenticated-root disable** command disabled the **Signed System Volume (SSV)**, which protects the System Volume. In a few words, all files have SHA-256 hashes stored in the file system metadata, and such hashes are used to check whether any file has been tampered with.
- Additionally, another hash set covered all directories (and the entire volume) to verify the integrity of the volume each time that macOS starts, and that is the reason you have seen the word “sealed” associated to System volume.
- You could have chosen another temporary directory to perform the procedure.

- The set of values “debug=0x44 kdp_match_name=en0 wdt=-1” are not really necessary here. They are used as a procedure for kernel debugging of a panic system (through NMI), using en0 as network communication and disabling watchdog monitoring. I will quickly mention them later.
- This procedure is precise and evaluated. However, if you are using a virtual machine, you can take a snapshot before it starts to be able to roll back the environment if something goes wrong.

To debug it, readers can follow the same steps:

1. **lldb**
2. (lldb) **target create /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel**
3. (lldb) **settings set target.load-script-from-symbol-file true**
4. (lldb) **gdb-remote 192.168.0.96:8864**

```
(lldb) gdb-remote 192.168.0.96:8864
Kernel UUID: E4D580D5-5966-3C46-8B24-5E7CE059C52D
Load Address: 0xffffff801e4f8000
Target arch: x86_64
Connected to live debugserver or arm core. Will assc
ate on-core threads to registers reported by server.
Process 1 stopped
* thread #1, stop reason = signal SIGTRAP
    frame #0: 0xffffff801e8728a4
->  0xffffff801e8728a4: cli
    0xffffff801e8728a5: pushfq
    0xffffff801e8728a6: popq   %rax
    0xffffff801e8728a7: testl  $0x200, %eax
   ; imm = 0x200
Target 0: (No executable module.) stopped.
(lldb) bt
* thread #1, stop reason = signal SIGTRAP
  * frame #0: 0xffffff801e8728a4
    frame #1: 0xffffff801e6f7833
    frame #2: 0xffffff801e6f7ae6
(lldb) f 2
frame #2: 0xffffff801e6f7ae6
->  0xffffff801e6f7ae6: cmpl   $0x0, %gs:0x58
    0xffffff801e6f7aef: jle    0xffffff801e6f7bcb
    0xffffff801e6f7af5: movq   %rax, %r14
    0xffffff801e6f7af8: cmpl   $0x0, 0x10538e1(%rip)
(lldb) register read
general:
       rbp = 0xffffffff001effa0
       rsp = 0xffffffff001eff90
       rip = 0xffffff801e6f7ae6
38 registers were unavailable.
```

```
(lldb) f 0
frame #0: 0xffffff801e8728a4
->  0xffffff801e8728a4: cli
    0xffffff801e8728a5: pushfq
    0xffffff801e8728a6: popq   %rax
    0xffffff801e8728a7: testl  $0x200, %eax
   ; imm = 0x200
(lldb) register read
general:
       rax = 0x000000000000041c
       rbx = 0x000000997f9dd3a3
       rcx = 0xffffff801f562c90
       rdx = 0x0000000000000000
       rsi = 0x0000000000000002
       rdi = 0xffffff801f584160
       rbp = 0xffffffff001eff20
       rsp = 0xffffffff001eff10
        r8 = 0x1234abcddcba4321
        r9 = 0x000000000000000f
       r10 = 0xffffff801f593700
       r11 = 0xffffff801f58b1d0
       r12 = 0xffffffff001eff40
       r13 = 0xffffff801f58b6e0
       r14 = 0xffffff801f508a40
       r15 = 0xffffffff001eff48
       rip = 0xffffff801e8728a4
    eflags = 0x00000246
        cs = 0x00000008
        ss = 0x00000010
        ds = 0x00000000
        es = 0x00000000
        fs = 0x00000000
        gs = 0x00000000
```

```
[(lldb) disassemble -f -b
 ->  0xffffff801e8728a4: fa                      cli
     0xffffff801e8728a5: 9c                      pushfq
     0xffffff801e8728a6: 58                      popq   %rax
     0xffffff801e8728a7: a9 00 02 00 00          testl  $0x200, %eax              ; imm =
     0xffffff801e8728ac: 0f 85 40 01 00 00       jne    0xffffff801e8729f2
     0xffffff801e8728b2: 65 48 8b 04 25 00 00 00 00 movq   %gs:0x0, %rax
     0xffffff801e8728bb: 48 83 a0 c0 00 00 00 fe     andq   $-0x2, 0xc0(%rax)
```

[Figure 23]: quick view of development kernel debugging session

To exit LLDB and leave the remote macOS kernel running, execute **detach** and **exit** commands, respectively.

We have established a way to debug the macOS kernel, but we can get better results including the source code. We should remember that Apple make the XNU source code, which is available on https://github.com/apple-oss-distributions/xnu. Therefore, we can clone this repository and choose the exact version (or almost…) of our XNU code, which can be verified by running **sysctl kern.version** command, which shows the following output:

> kern.version: Darwin Kernel Version 23.6.0: Mon Jul 29 21:13:00 PDT 2024**; root:xnu-10063.141.2**~1/RELEASE_X86_64

We already know what our XNU version is and now we can retrieve it. Here things can go for different approaches, using commands such as **git** (old and new syntax) and also **gh** (GitHub CLI) commands, but I will try to do the most basic and well-known procedure, but readers can take their own way.

```
alexandreborges@Mac github % git clone https://github.com/apple-oss-distributions/xnu
Cloning into 'xnu'...
remote: Enumerating objects: 64181, done.
remote: Counting objects: 100% (10138/10138), done.
remote: Compressing objects: 100% (1419/1419), done.
remote: Total 64181 (delta 8673), reused 10098 (delta 8668), pack-reused 54043 (from 1)
Receiving objects: 100% (64181/64181), 117.78 MiB | 18.39 MiB/s, done.
Resolving deltas: 100% (53153/53153), done.
Updating files: 100% (5187/5187), done.
alexandreborges@Mac github %
alexandreborges@Mac github % cd xnu
alexandreborges@Mac xnu % git fetch
alexandreborges@Mac xnu % git branch -v -a
* main                          94d3b452 xnu-10063.101.15
  remotes/origin/HEAD           -> origin/main
  remotes/origin/main           94d3b452 xnu-10063.101.15
  remotes/origin/rel/xnu-10002 5e3eaea3 xnu-10002.81.5
  remotes/origin/rel/xnu-10063 d8b80295 xnu-10063.141.1
  remotes/origin/rel/xnu-1228  08333225 xnu-1228.15.4
  remotes/origin/rel/xnu-124   6681e14e xnu-124.13
  remotes/origin/rel/xnu-1504  99c11f3f xnu-1504.15.3
  remotes/origin/rel/xnu-1699  161efb48 xnu-1699.32.7
  remotes/origin/rel/xnu-201   1c36fc8e xnu-201.42.3
  remotes/origin/rel/xnu-2050  4abd0e59 xnu-2050.48.11
  remotes/origin/rel/xnu-2422  d2a0abf2 xnu-2422.115.4
  remotes/origin/rel/xnu-2782  c2b85efb xnu-2782.40.9
  remotes/origin/rel/xnu-3247  1db20409 xnu-3247.10.11
  remotes/origin/rel/xnu-3248  bf8b1712 xnu-3248.60.10
  remotes/origin/rel/xnu-344   0554d6e8 xnu-344.49
  remotes/origin/rel/xnu-3789  d0030a38 xnu-3789.70.16
  remotes/origin/rel/xnu-4570  925687e1 xnu-4570.71.2
  remotes/origin/rel/xnu-4903  494ffe1c xnu-4903.270.47
  remotes/origin/rel/xnu-517   0a8e6516 xnu-517.12.7
  remotes/origin/rel/xnu-6153  5cb76f88 xnu-6153.141.1
  remotes/origin/rel/xnu-7195  776661b7 xnu-7195.141.2
  remotes/origin/rel/xnu-792   432bdb7e xnu-792.25.20
  remotes/origin/rel/xnu-8019  a325d9c4 xnu-8019.80.24
  remotes/origin/rel/xnu-8020  27b03b36 xnu-8020.140.41
  remotes/origin/rel/xnu-8792  19c3b8c2 xnu-8792.81.2
  remotes/origin/rel/xnu-8796  1b191cb5 xnu-8796.141.3
```

[Figure 24]: Cloning and listing existing branches

```
alexandreborges@Mac xnu % git switch -c xnu-10063 origin/rel/xnu-10063
branch 'xnu-10063' set up to track 'origin/rel/xnu-10063'.
Switched to a new branch 'xnu-10063'
alexandreborges@Mac xnu % git branch -a -v
  main                          94d3b452 xnu-10063.101.15
* xnu-10063                     d8b80295 xnu-10063.141.1
  remotes/origin/HEAD           -> origin/main
  remotes/origin/main           94d3b452 xnu-10063.101.15
  remotes/origin/rel/xnu-10002 5e3eaea3 xnu-10002.81.5
  remotes/origin/rel/xnu-10063 d8b80295 xnu-10063.141.1
  remotes/origin/rel/xnu-1228  08333225 xnu-1228.15.4
  remotes/origin/rel/xnu-124   6681e14e xnu-124.13
  remotes/origin/rel/xnu-1504  99c11f3f xnu-1504.15.3
  remotes/origin/rel/xnu-1699  161efb48 xnu-1699.32.7
  remotes/origin/rel/xnu-201   1c36fc8e xnu-201.42.3
  remotes/origin/rel/xnu-2050  4abd0e59 xnu-2050.48.11
  remotes/origin/rel/xnu-2422  d2a0abf2 xnu-2422.115.4
  remotes/origin/rel/xnu-2782  c2b85efb xnu-2782.40.9
  remotes/origin/rel/xnu-3247  1db20409 xnu-3247.10.11
  remotes/origin/rel/xnu-3248  bf8b1712 xnu-3248.60.10
  remotes/origin/rel/xnu-344   0554d6e8 xnu-344.49
  remotes/origin/rel/xnu-3789  d0030a38 xnu-3789.70.16
  remotes/origin/rel/xnu-4570  925687e1 xnu-4570.71.2
  remotes/origin/rel/xnu-4903  494ffe1c xnu-4903.270.47
  remotes/origin/rel/xnu-517   0a8e6516 xnu-517.12.7
  remotes/origin/rel/xnu-6153  5cb76f88 xnu-6153.141.1
  remotes/origin/rel/xnu-7195  776661b7 xnu-7195.141.2
  remotes/origin/rel/xnu-792   432bdb7e xnu-792.25.20
  remotes/origin/rel/xnu-8019  a325d9c4 xnu-8019.80.24
  remotes/origin/rel/xnu-8020  27b03b36 xnu-8020.140.41
  remotes/origin/rel/xnu-8792  19c3b8c2 xnu-8792.81.2
  remotes/origin/rel/xnu-8796  1b191cb5 xnu-8796.141.3
```

[Figure 25]: Switching to the target branch (xnu-10063.141.1)

As readers can notice from the code above, it is not the exact same branch, but it is pretty close. Now we can try opening a remote debugging session again, but using the XNU source code and also applying other slight changes:

1. **lldb**
2. **settings set target.x86-disassembly-flavor intel**
3. **settings set target.source-map /Library/Caches/com.apple.xbs/Sources/xnu/xnu-10063.141.2 /Users/alexandreborges/github/xnu-10063.141.2**
4. **command script import "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Resources/Python/lldbmacros/xnu.py"**
5. **command script import "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Resources/Python/lldbmacros/memory.py"**
6. **target create /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.development**
7. **settings set target.load-script-from-symbol-file true**
8. **gdb-remote 192.168.0.96:8864**

Once again, a few words about different commands above:

- **command 1:** it launches LLDB.
- **command 2**: it changes the assembly representation to Intel, which is much better.
- **command 3:** it sets the source code location based on our previous cloned code.
- **command 4 and 5:** the script is loaded when the target is set just in case both are not loaded.
- **command 6:** it loads the target kernel that will be debugged. Take care: you must run **sysctl kern.version** to check whether the system is running either release or development kernel.
- **command 7:** it runs all discovered debug scripts in this section.
- **command 8:** it opens the remote debugging session.

Of course, if readers do not want to waste time typing all these commands again, you should create a file under **~/.lldb** directory as shown below:

```
alexandreborges@Mac .lldb % pwd
/Users/alexandreborges/.lldb
alexandreborges@Mac .lldb %
alexandreborges@Mac .lldb % cat remote_debugging
settings set target.x86-disassembly-flavor intel
settings set target.source-map /Library/Caches/com.apple.xbs/Sources/xnu/xnu-10063.141.2
/Users/alexandreborges/github/xnu-10063.141.2
command script import "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernel
s/kernel.development.dSYM/Contents/Resources/Python/lldbmacros/xnu.py"
command script import "/Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernel
s/kernel.development.dSYM/Contents/Resources/Python/lldbmacros/memory.py"
settings set target.load-script-from-symbol-file true
target create /Library/Developer/KDKs/KDK_14.6.1_23G93.kdk/System/Library/Kernels/kernel.
development
gdb-remote 192.168.0.96:8864
```

[Figure 26]: Remote debugging script

Now we have to launch the LLDB (lldb command) and execute the script as shown below:

- **lldb**
- **command source /Users/alexandreborges/.lldb/remote_debugging**

```
Process 1 stopped
* thread #1, stop reason = signal SIGTRAP
    frame #0: 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3 [opt]
Target 0: (kernel.development) stopped.
(lldb) disassemble
kernel.development`machine_idle:
    0xffffff80006726d0 <+0>:   push   rbp
    0xffffff80006726d1 <+1>:   mov    rbp, rsp
    0xffffff80006726d4 <+4>:   push   r14
    0xffffff80006726d6 <+6>:   push   rbx
    0xffffff80006726d7 <+7>:   mov    r14, qword ptr gs:[0x0]
    0xffffff80006726e0 <+16>:  lea    rdi, [rip + 0xd11a79]     ; pal_rtc_nanotime_info
    0xffffff80006726e7 <+23>:  call   0xffffff8000447110        ; _rtc_nanotime_read
    0xffffff80006726ec <+28>:  mov    rbx, rax
    0xffffff80006726ef <+31>:  cmp    rax, qword ptr [r14 + 0x78]
    0xffffff80006726f3 <+35>:  jb     0xffffff8000672731        ; <+97> at pmCPU.c:131:21
    0xffffff80006726f5 <+37>:  mov    rax, qword ptr [r14 + 0x98]
    0xffffff80006726fc <+44>:  sub    rax, rbx
    0xffffff80006726ff <+47>:  jbe    0xffffff8000672731        ; <+97> at pmCPU.c:131:21
    0xffffff8000672701 <+49>:  mov    ecx, dword ptr [rip + 0xd11e01] ; idle_entry_timer_
```

[Figure 27]: Assembly with references to the source code

```
(lldb) thread list
Process 1 stopped
* thread #1: tid = 0x0001, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3,
stop reason = signal SIGTRAP
  thread #2: tid = 0x0002, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3
  thread #3: tid = 0x0003, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3
  thread #4: tid = 0x0004, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3
(lldb) bt
* thread #1, stop reason = signal SIGTRAP
  * frame #0: 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3 [opt]
    frame #1: 0xffffff80004f7833 kernel.development`processor_idle(thread=0x0000000000000000, pr
ocessor=0xffffff800138b1d0) at sched_prim.c:6669:3 [opt]
    frame #2: 0xffffff80004f7ae6 kernel.development`idle_thread(parameter=<unavailable>, result=
<unavailable>) at sched_prim.c:6766:24 [opt]
    frame #3: 0xffffff800044719e kernel.development`call_continuation + 46
(lldb) frame info
frame #0: 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179:3 [opt]
(lldb) f 2
frame #2: 0xffffff80004f7ae6 kernel.development`idle_thread(parameter=<unavailable>, result=<una
vailable>) at sched_prim.c:6766:24 [opt]
(lldb) frame info
frame #2: 0xffffff80004f7ae6 kernel.development`idle_thread(parameter=<unavailable>, result=<una
vailable>) at sched_prim.c:6766:24 [opt]
(lldb) register read rax rcx rsp rbp
rax           = error: unavailable
rcx           = error: unavailable
     rsp = 0xffffffba826f7f90
     rbp = 0xffffffba826f7fa0
(lldb) di -l


kernel.development`idle_thread:
    0xffffff80004f7adc <+140>: xor    edi, edi
    0xffffff80004f7ade <+142>: mov    rsi, rbx
    0xffffff80004f7ae1 <+145>: call   0xffffff80004f76d0        ; processor_idle at sched_prim.c
:6614
(lldb) b mmap
Breakpoint 2: where = kernel.development`mmap + 47 at kern_mman.c:189:19, address = 0xffffff
f8000a2b3bf
(lldb) br l
Current breakpoints:
2: name = 'mmap', locations = 1, resolved = 1, hit count = 0
  2.1: where = kernel.development`mmap + 47 at kern_mman.c:189:19, address = 0xffffff8000a2
b3bf, resolved, hit count = 0

(lldb) c
Process 1 resuming
Process 1 stopped
* thread #8, name = '0xffffffa07acbb188', queue = 'cpu-3', stop reason = breakpoint 2.1
    frame #0: 0xffffff8000a2b3bf kernel.development`mmap(p=0xffffffa07bd89038, uap=0xffffff
a07acbb980, retval=0xffffffa07acbb9c0) at kern_mman.c:189:19 [opt]
Target 0: (kernel.development) stopped.
(lldb) di -f -c 20
kernel.development`mmap:
    0xffffff8000a2b390 <+0>:  push   rbp
    0xffffff8000a2b391 <+1>:  mov    rbp, rsp
    0xffffff8000a2b394 <+4>:  push   r15
    0xffffff8000a2b396 <+6>:  push   r14
    0xffffff8000a2b398 <+8>:  push   r13
    0xffffff8000a2b39a <+10>: push   r12
    0xffffff8000a2b39c <+12>: push   rbx
```

```
    0xffffff8000a2b39d <+13>: sub    rsp, 0x288
    0xffffff8000a2b3a4 <+20>: mov    r15, rdx
    0xffffff8000a2b3a7 <+23>: mov    r12, rsi
    0xffffff8000a2b3aa <+26>: mov    qword ptr [rbp - 0x288], rdi
    0xffffff8000a2b3b1 <+33>: lea    rax, [rip + 0x962c88]     ; __stack_chk_guard
    0xffffff8000a2b3b8 <+40>: mov    rax, qword ptr [rax]
    0xffffff8000a2b3bb <+43>: mov    qword ptr [rbp - 0x30], rax
->  0xffffff8000a2b3bf <+47>: int3
    0xffffff8000a2b3c0 <+48>: mov    dword ptr [rbp - 0x260], 0x0
    0xffffff8000a2b3ca <+58>: mov    dword ptr [rbp - 0x1fc], 0x0
    0xffffff8000a2b3d4 <+68>: movsxd r13, dword ptr [rsi + 0x20]
    0xffffff8000a2b3d8 <+72>: mov    rbx, qword ptr gs:[0x10]
    0xffffff8000a2b3e1 <+81>: mov    r8, qword ptr [rbx + 0x6b0]

(lldb) thread list
Process 1 stopped
  thread #5: tid = 0x0066, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179
:3, name = '0xffffffa07bd5a3d0', queue = 'cpu-0'
  thread #6: tid = 0x00a3, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179
:3, name = '0xffffffa07bcee7a0', queue = 'cpu-1'
  thread #7: tid = 0x00a6, 0xffffff80006728a4 kernel.development`machine_idle at pmCPU.c:179
:3, name = '0xffffffa07bcefdb8', queue = 'cpu-2'
* thread #8: tid = 0x0d6f, 0xffffff8000a2b3bf kernel.development`mmap(p=0xffffffa07bd89038,
uap=0xffffffa07acbb980, retval=0xffffffa07acbb9c0) at kern_mman.c:189:19, name = '0xffffffa0
7acbb188', queue = 'cpu-3', stop reason = breakpoint 2.1
(lldb) thread backtrace
* thread #8, name = '0xffffffa07acbb188', queue = 'cpu-3', stop reason = breakpoint 2.1
  * frame #0: 0xffffff8000a2b3bf kernel.development`mmap(p=0xffffffa07bd89038, uap=0xffffffa
07acbb980, retval=0xffffffa07acbb9c0) at kern_mman.c:189:19 [opt]
    frame #1: 0xffffff8000bd42b8 kernel.development`unix_syscall64(state=0xffffffa07ad5ce20)
 at systemcalls.c:386:10 [opt]
    frame #2: 0xffffff8000447db6 kernel.development`hndl_unix_scall64 + 22
(lldb) f 1
frame #1: 0xffffff8000bd42b8 kernel.development`unix_syscall64(state=0xffffffa07ad5ce20) at
systemcalls.c:386:10 [opt]
(lldb) frame variable
(x86_saved_state_t *) state = 0xffffffa07ad5ce20
(thread_t) thread = <variable not available>

(void *) vt = 0xffffffa07acbb980
(unsigned int) code = 197
(unsigned int) syscode = 197
(const sysent *) callp = 0xffffff8000253168
(int) args_in_regs = <variable not available>

(boolean_t) args_start_at_rdi = <variable not available>

(int) error = <variable not available>

(proc *) p = 0xffffffa07bd89038
(uthread *) uthread = 0xffffffa07acbb980
(x86_saved_state64_t *) regs = <variable not available>

(pid_t) pid = 487
(kern_allocation_name_t) prior = <variable not available>
```

[Figure 28]: Overview about LLDB

I have shown basic commands on LLDB (only as an overview), and they are similar to GDB, so I do not need to explain they are pretty obvious. Try to type **kgmhelp** and you will have an extensive list of XNU commands.

For reverting all changes and return the kernel release version, we must perform the following steps:

- **sudo nvram -d boot-args**
- identify the system volume disk device name, which is mounted at "/": **mount | grep sealed**. In my case, it is /dev/disk1s4
- (only if we deleted from the previous procedure) create a temporary mount directory: **mkdir /Users/ab/livemount** (as it is usually done).
- mount the system volume onto the temporary directory: **sudo mount -o nobrowse -t apfs /dev/disk1s4 /Users/ab/livemount**
- bless (authorize) the restore booting from the previously sealed snapshot: **sudo bless --mount /Users/ab/livemount --bootefi --last-sealed-snapshot**
- If we want, we can also reverse the SIP and authenticated root configuration through the Recovery Mode:
  - **csrutil authenticated-root disable**
  - **csrutil enable**
- Reboot the system: **sudo reboot**
- Check variant using **sw_vers** and you will see *"kernel.osbuildconfig: release"* as the output.

There is another way to establish a debugging section using **KDP (Kernel Debugging Protocol)**, but it is more interesting for people using physical machines (Intel-based machines and not Silicon machines) than virtual machines. This procedure below assumes you already disabled SIP and **Signed System Volume (SSV)**, as shown previously and if you check the XNU source code (**xnu\bsd\sys\csr.h**) you can learn about what is involved while you enable or disable SIP, and other quite interesting details:

```
41: /* CSR configuration flags */
42: #define CSR_ALLOW_UNTRUSTED_KEXTS               (1 << 0)
43: #define CSR_ALLOW_UNRESTRICTED_FS               (1 << 1)
44: #define CSR_ALLOW_TASK_FOR_PID                  (1 << 2)
45: #define CSR_ALLOW_KERNEL_DEBUGGER               (1 << 3)
46: #define CSR_ALLOW_APPLE_INTERNAL                (1 << 4)
47: #define CSR_ALLOW_DESTRUCTIVE_DTRACE                    (1 << 5) /* name deprecated */
48: #define CSR_ALLOW_UNRESTRICTED_DTRACE                   (1 << 5)
49: #define CSR_ALLOW_UNRESTRICTED_NVRAM                    (1 << 6)
50: #define CSR_ALLOW_DEVICE_CONFIGURATION                  (1 << 7)
51: #define CSR_ALLOW_ANY_RECOVERY_OS                       (1 << 8)
52: #define CSR_ALLOW_UNAPPROVED_KEXTS                      (1 << 9)
53: #define CSR_ALLOW_EXECUTABLE_POLICY_OVERRIDE    (1 << 10)
54: #define CSR_ALLOW_UNAUTHENTICATED_ROOT                  (1 << 11)
55:
56: #define CSR_VALID_FLAGS (CSR_ALLOW_UNTRUSTED_KEXTS | \
57:                          CSR_ALLOW_UNRESTRICTED_FS | \
58:                          CSR_ALLOW_TASK_FOR_PID | \
59:                          CSR_ALLOW_KERNEL_DEBUGGER | \
60:                          CSR_ALLOW_APPLE_INTERNAL | \
61:                          CSR_ALLOW_UNRESTRICTED_DTRACE | \
62:                          CSR_ALLOW_UNRESTRICTED_NVRAM | \
63:                          CSR_ALLOW_DEVICE_CONFIGURATION | \
64:                          CSR_ALLOW_ANY_RECOVERY_OS | \
65:                          CSR_ALLOW_UNAPPROVED_KEXTS | \
66:                          CSR_ALLOW_EXECUTABLE_POLICY_OVERRIDE | \
67:                          CSR_ALLOW_UNAUTHENTICATED_ROOT)
68:
69: #define CSR_ALWAYS_ENFORCED_FLAGS (CSR_ALLOW_DEVICE_CONFIGURATION | CSR_ALLOW_ANY_RECOVERY_OS)
70:
71: /* Flags set by `csrutil disable`. */
72: #define CSR_DISABLE_FLAGS (CSR_ALLOW_UNTRUSTED_KEXTS | \
73:                            CSR_ALLOW_UNRESTRICTED_FS | \
74:                            CSR_ALLOW_TASK_FOR_PID | \
75:                            CSR_ALLOW_KERNEL_DEBUGGER | \
76:                            CSR_ALLOW_APPLE_INTERNAL | \
77:                            CSR_ALLOW_UNRESTRICTED_DTRACE | \
78:                            CSR_ALLOW_UNRESTRICTED_NVRAM)
```

[Figure 29]: xnu\bsd\sys\csr.h from XNU source code

Furthermore, the step-by-step procedure also assumes that system will be using kernel development version, which is a small detail because you might adapt nvram **boot-args variable** to kernel release version:

1. (target) Run **ifconfig** command and determine the active network device (most of the time, en0) and respective IP address (it is recommended configure it as a fixed IP address).
2. (target) Execute: **sudo nvram boot-args="debug=0x8C44 kdp_match_name=en0 wdt=-1 kcsuffix=development".** These parameters have the following meaning:
    a. **debug=0x8C44** – it configures the kernel to wait for a debugger to attach to the system, and more. Details are coming later.
    b. **kdp_match_name=en0** – it is the value of the network interface.
    c. **wdt=-1** – disables watchdog monitoring
3. (target) Reboot the target system: **sudo reboot**
4. (host) Install Xcode and the two Python packages running the following commands (I have mentioned it previously):
    a. xcode-select --install
    b. xcrun python3 -m pip install --user --ignore-installed macholib
    c. xcrun python3 -m pip install --user --ignore-installed future
5. (target) **Generate an NMI** (Non-Maskable Interrupt). I will talk about it soon after this procedure.
6. (host) **lldb**
7. (host) **target create /Library/Developer/KDKs/KDK_14.5_23F5074a.kdk/System/Library/Kernels/kernel.development**
8. (host) **kdp-remote 192.168.0.34** *(target IP address)*

This procedure works well on physical Apple machines, but readers can have a challenging time trying to do it on virtual machines because it is not quite trivial to trigger an NMI.

Apple offers a documentation (https://developer.apple.com/documentation/kernel/generating_a_non-maskable_interrupt) showing different methods to generate an NMI:

Mac	Key presses
Apple silicon	Command (Left) + Command (Right) + Delete
Intel-based Mac (with Apple T2 Security Chip)	Command (Left) + Command (Right) + Delete
Intel-based Mac (without Apple T2 Security Chip)	Command (Left) + Command (Right) + Power Button

[Figure 30]: How to trigger NMI (credits: Apple)

There are other options such as pressing **command + control + shift + option + escape** keys at the same time to break the host in the debugger or even using DTrace to get the same result**: sudo dtrace -w -n "BEGIN { panic(); }"**. If you are using a non-Apple keyboard, you will have problems in pressing all these keys at the same time, but you could remap this sequence to a simpler key set whether you are running virtual machines on VMware Fusion, for example. Regardless of the methods you want to use, the pivotal point is that this procedure using KDP is more useful and intrinsically related to physical machines than virtual machines, and you are using virtual machines, so you should prefer the approach using GDB server

from VMware. The debug parameter in the nvram **boot-args variable** has not been explained yet, but all possible values are exposed on **xnu\osgmk\kern\debug.h** file from the XNU source code:

```
/* Debug boot-args */
#define DB_HALT         0x1
//#define DB_PRT          0x2 -- obsolete
#define DB_NMI          0x4
#define DB_KPRT         0x8
#define DB_KDB          0x10
#define DB_ARP          0x40
#define DB_KDP_BP_DIS   0x80
//#define DB_LOG_PI_SCRN  0x100 -- obsolete
#define DB_KDP_GETC_ENA 0x200

#define DB_KERN_DUMP_ON_PANIC           0x400 /* Trigger core dump on panic*/
#define DB_KERN_DUMP_ON_NMI             0x800 /* Trigger core dump on NMI */
#define DB_DBG_POST_CORE                0x1000 /*Wait in debugger after NMI core */
#define DB_PANICLOG_DUMP                0x2000 /* Send paniclog on panic,not core*/
#define DB_REBOOT_POST_CORE             0x4000 /* Attempt to reboot after
                                            * post-panic crashdump/paniclog
                                            * dump.
                                            */
#define DB_NMI_BTN_ENA          0x8000  /* Enable button to directly trigger NMI */
/* 0x10000 was DB_PRT_KDEBUG (kprintf kdebug events), feature removed */
#define DB_DISABLE_LOCAL_CORE   0x20000 /* ignore local kernel core dump support */
#define DB_DISABLE_GZIP_CORE    0x40000 /* don't gzip kernel core dumps */
#define DB_DISABLE_CROSS_PANIC  0x80000 /* x86 only - don't trigger cross panics. Only
                                        * necessary to enable x86 kernel debugging on
                                        * configs with a dev-fused co-processor running
                                        * release bridgeOS.
                                        */
#define DB_REBOOT_ALWAYS        0x100000 /* Don't wait for debugger connection */
#define DB_DISABLE_STACKSHOT_TO_DISK 0x200000 /* Disable writing stackshot to local disk */
#define DB_DEBUG_IP_INIT        0x400000 /* iBoot specific: Allow globally enabling debug IPs during init */
#define DB_SOC_HALT_ENABLE      0x800000 /* iBoot specific: Enable SoC Halt during init */
```

[Figure 31]: XNU debug boot-args

Refreshing about our prior choice on the last page, we used the value 0x8C44, which enables button to directly trigger NMI, trigger core dump on panic and NMI, enables communication in the private network using ARP and enables NMI. Of course, other combinations of values are possible. This debugging procedure has limitations that, for now, it is not possible to actively debugging the kernel of an Apple silicon machine, and according to Kernel Debug Kit for macOS - Read Me document (written by Apple):

> *"On Apple silicon, perform two-machine debugging using your Mac's built-in Ethernet ports.*
>
> *On Intel-based Macs, perform two-machine debugging using your Mac's built-in Ethernet ports, the Ethernet port on the Apple Thunderbolt display, or the Apple Thunderbolt to Gigabit Ethernet adapter*
>
> *(…)*
>
> *Note: You cannot perform two-machine debugging using USB Ethernet adapters or wireless networking on any Mac.*
>
> *You must connect the host and target device to the same network, but there are no other restrictions on how the devices connect to that network."*

According to the explanations given by Apple, we cannot use USB Ethernet and wireless adapters. Therefore:

- Apple silicon machines do not support active and can only analyze crashes.
- We need Thunderbolt adapters.

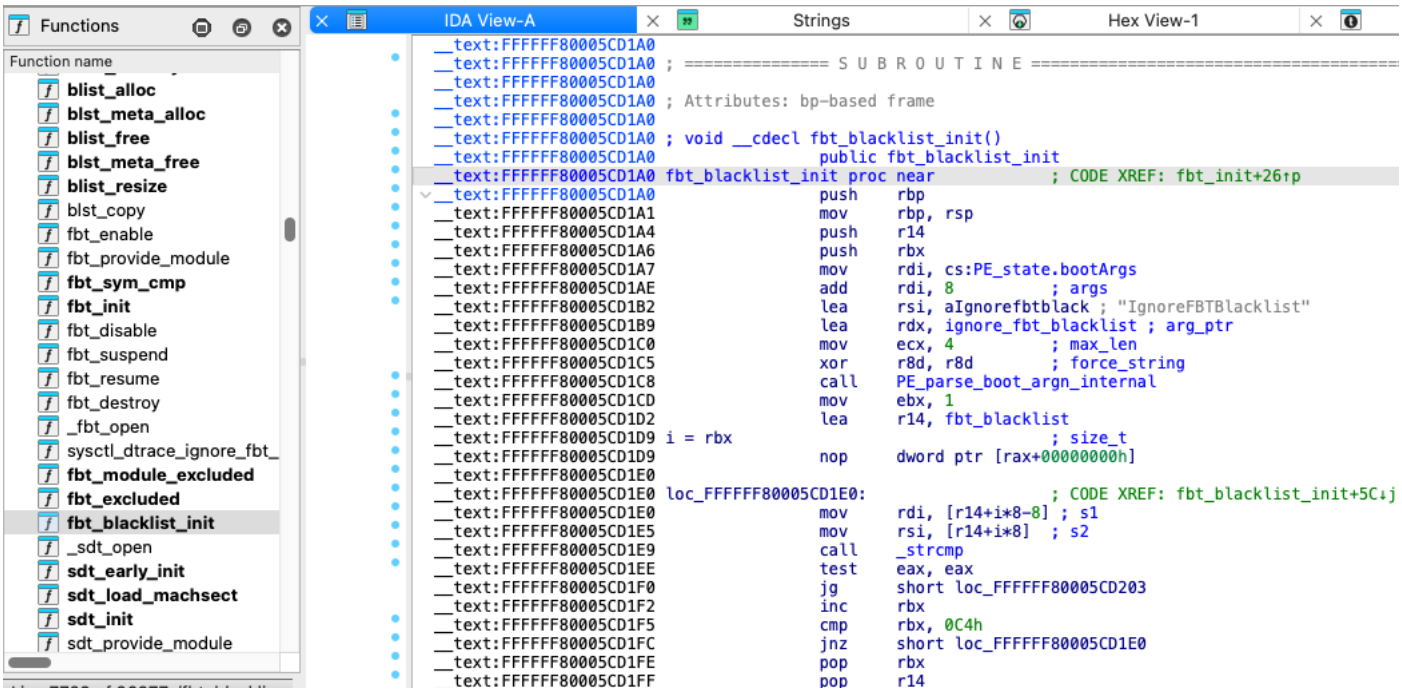
In my case, I have a MacBook M2 machine, which came with a Thunderbolt v4 port, and two adapters are necessary (both sold by Apple):

- Thunderbolt 3 (USB-C) to Thunderbolt 2 Adapter
- Belkin USB-C to Gigabit Ethernet Adapter

Nonetheless, as we are interested in performing active kernel debugging, the core/crash dump analysis will not be presented.

06. XNU kernel debugging using IDA Pro

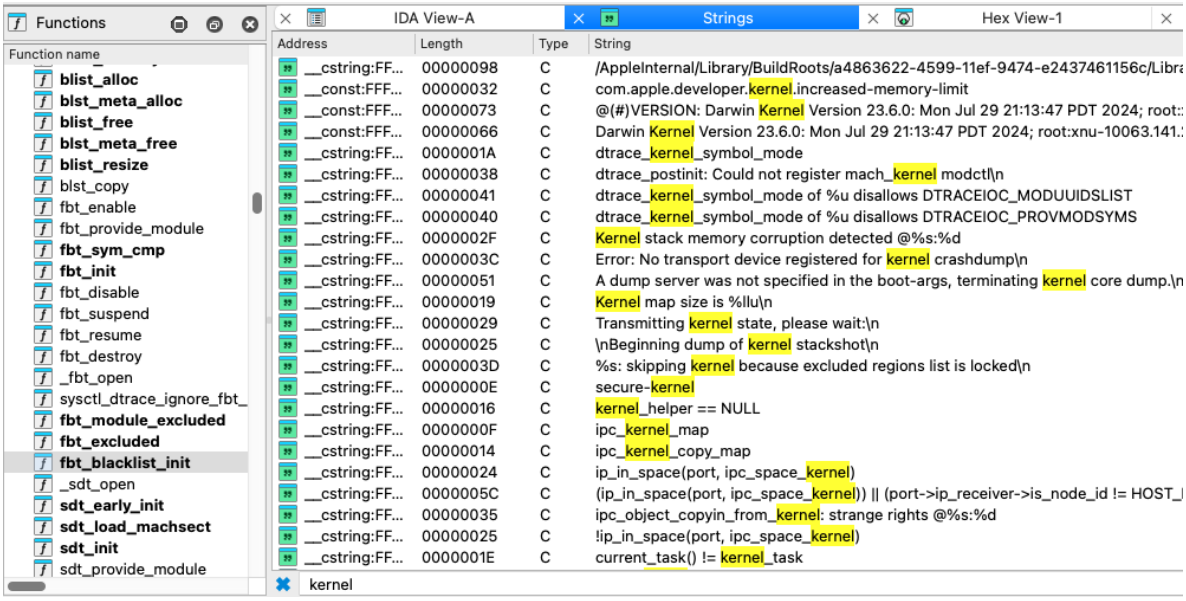
If you are curious about the content of the kernel.development file, open it on IDA Pro (you must be using one IDA version able to open Mach-O format), and you will see that there are many function names (symbols) and strings, as shown below:



[Figure 32]: kernel.development and its respective IDA Pro disassembly view and functions

As we can see, there is a staggering number of functions, and it is expected because we are talking about a kernel like XNU. However, we have access to the respective DWARF files, so also a respectable number of type definitions (**View > Open subviews > Local Types**). If you want (and would be not strictly necessary because we have access to the XNU source code), you can generate a full decompiled file by navigating to **File > Produce File > Create C File…** .This action takes some time. Using Lumina (menu **Lumina > Pull all**) bring some light on a few functions and, as I always say, every help is always welcome.

If readers check **Strings (View > Open subviews > Strings**) you will see many strings available in this version of **kernel.development**, and can use them as a guideline to analyze essential functions and mechanisms:



[Figure 33]: kernel.development and its respective strings

Right now, we are not interested in reversing statically the macOS kernel, but learning how to debug XNU kernel using a tool such as IDA Pro. One of the first things is establishing the appropriate arguments to boot the macOS, and we are going to use:

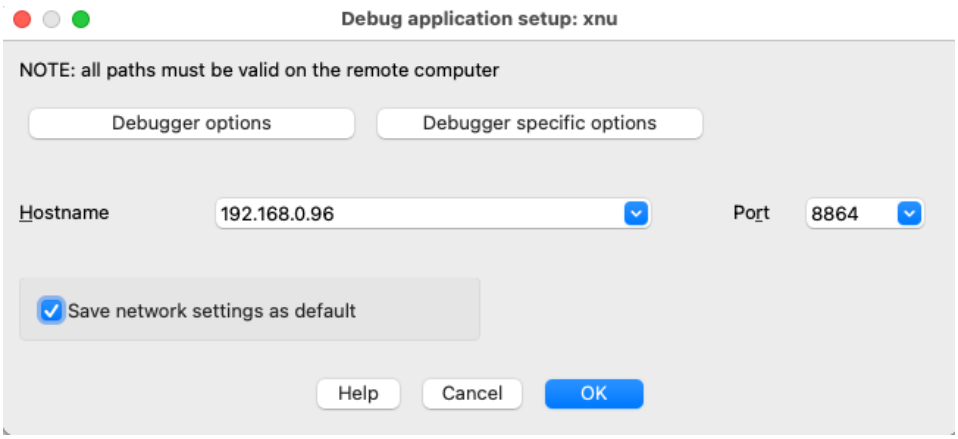
- **slide=0:** it disables KASLR (Kernel Address Space Layout Randomization).
- **bootsyms:** it enables you to display all possible kernel symbols during the boot process.
- **wdt=1:** it enables watchdog timer, which resets the system whether it becomes unresponsive.

Therefore, by reusing the previous boot-args, we can build the following command:

- **sudo nvram boot-args="slide=0 debug=0x8C44 kdp_match_name=en0 wdt=-1 keepsyms=1 kcsuffix=development"**

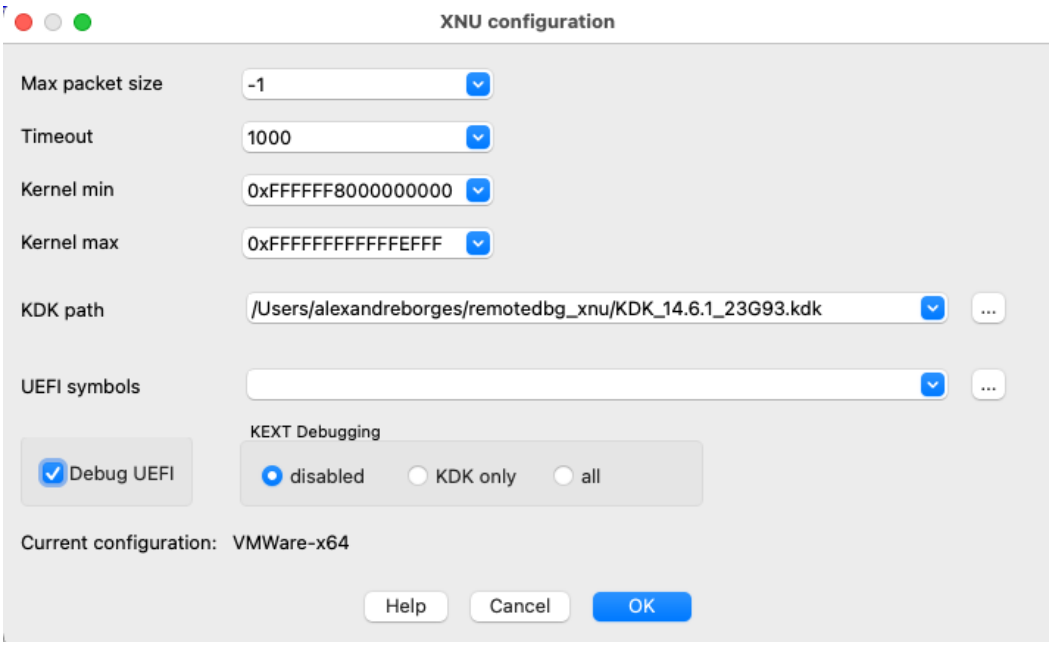
Although I am not going to deploy KDP, I left here because it does not cause any issue because debugging will take advantage of VMware GDB from previous sections. As usual, we need to reboot: **sudo reboot**

On the host, launch the IDA Pro with an empty database (**/Applications/IDA Pro 8.4/idabin/ida64 -t**), to **Debugger > Attach > Remote XNU debugger**, and set the following options:



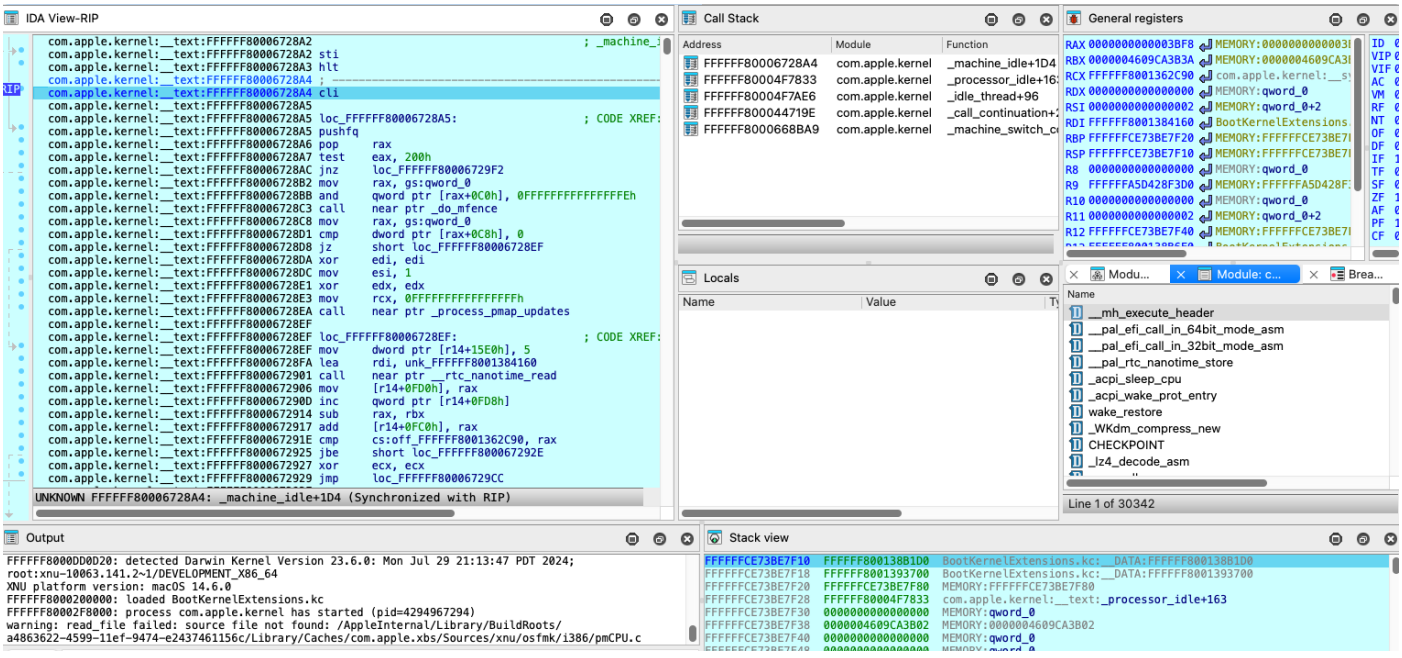
[Figure 34]: IDA Pro XNU debugger configuration

Go to **Debugger Specific options** and check the configuration. Depending on the context, your available configurations can be different, but in my case **VMware-x64** and **Corellium-ARM64** are available and, as I have used VMware, the choice is obvious:



[Figure 35]: IDA Pro XNU debugger configuration

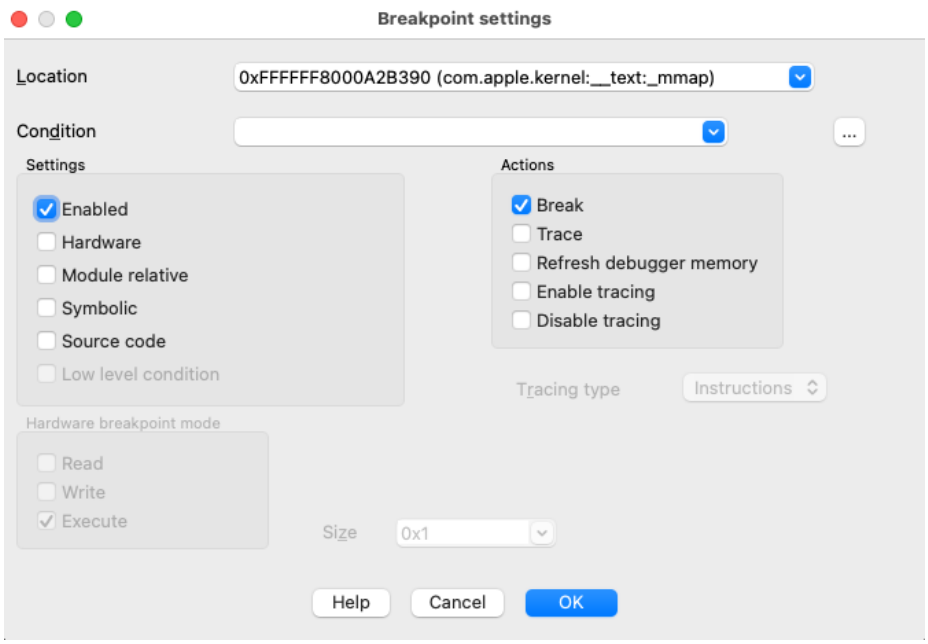
Confirming the configuration above and proceeding with **OK** on the **Debug application setup: xnu** window, you will see other three prompts whose only option will be to press **OK**, and then:



[Figure 36]: IDA Pro XNU debugger configuration

That is worked! We are debugging the macOS XNU kernel remotely and, similar to Windows, we can investigate internals mechanisms. We provide the KDK folder, which we have already downloaded previously, and which improves the debugging experience. There are a few notes:

- I have copied the KDK to a separate directory (**remotedbg_xnu**) on the home directory.
- The .idb file (**kernel_debugging.i64**) is saved on my home directory.
- Add the Locals view (**Debugger > Debugger windows > Locals**) as I did on the previous page to observe the possible associated structures (types). At the same way, I also suggest adding the **Call Stack view** (**Debugger > Debugger windows > Stack Trace**)
- If something went wrong, check if you really rebooted the system using the correct boot-args parameter: **nvram -p | grep boot-args**
- Check the kernel and its respective version using one or more of the following commands:
  - **sysctl kern.version**
  - **uname -a**
  - **system_profiler SPSoftwareDataType | grep "Kernel Version"**
- If you are facing instability over the debugging session, go to **Options** > **Source paths… menu** and **unmark Show this dialog when a source file cannot be found**, and press **Apply button**.
- If you want, you can evaluate breakpoints and, to accomplish this task, double-click the com.apple.kernel module, and pick up a few functions. For each function**, right-click it and choose Add Breakpoint**:



[Figure 37]: IDA Pro XNU debugger configuration

- A few examples of functions follow below:
  - _mmap
  - _ipc_mqueue_receive
  - _vm_fault
  - _kalloc
  - __mac_syscall

- To disable, go to the Breakpoint view (**Debugger > Breakpoint list**), right click the breakpoint, and choose **Disable breakpoint**.

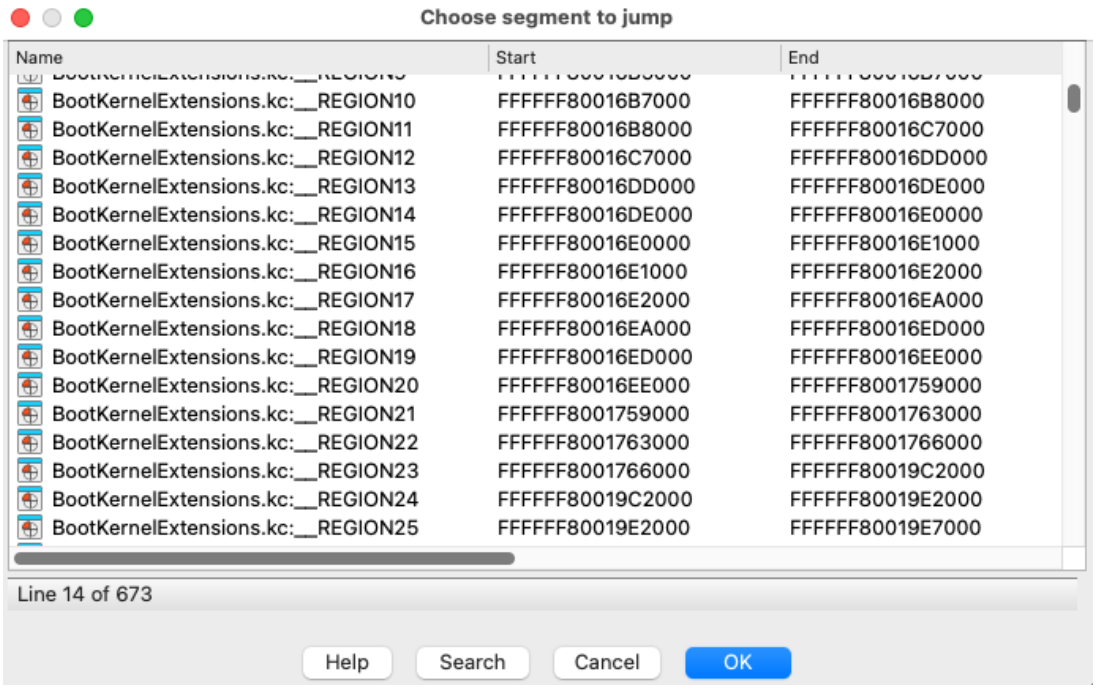
- If you are facing problems with breakpoints, try to change the respective setting in the virtual machine **.vmx file** by changing **hideBreakpoints** to “FALSE”. Of course, you need to stop the virtual machine, edit the .vmx, boot the macOS again and start the debugging session:
  - **debugStub.hideBreakpoints** = "FALSE"
- To allow the debugee (target) to continue without debugging, go to **Debugger > detach from process** and then press **No**.

In previous versions of macOS (before Big Sur), we could use pre-linked kernelcache to improve our information for debugging, but Apple has used kernel collections (**/System/Library/Extensions folder**), which are managed by **kmutil** command. For example, you can list each extension by running the **kmutil find** command.

On IDA Pro you should see two modules, at least:

- **com.apple.kernel**
- **BootKernelExtensions.kc**

The com.apple.kernel is a subset from **BootKernelExtension.kc**, and the multiple **BootKernelExtension.ks segments** (view by pressing **SHIFT+S**) represent the kext subfile, which are referred to and loaded, but not detected by debugger:

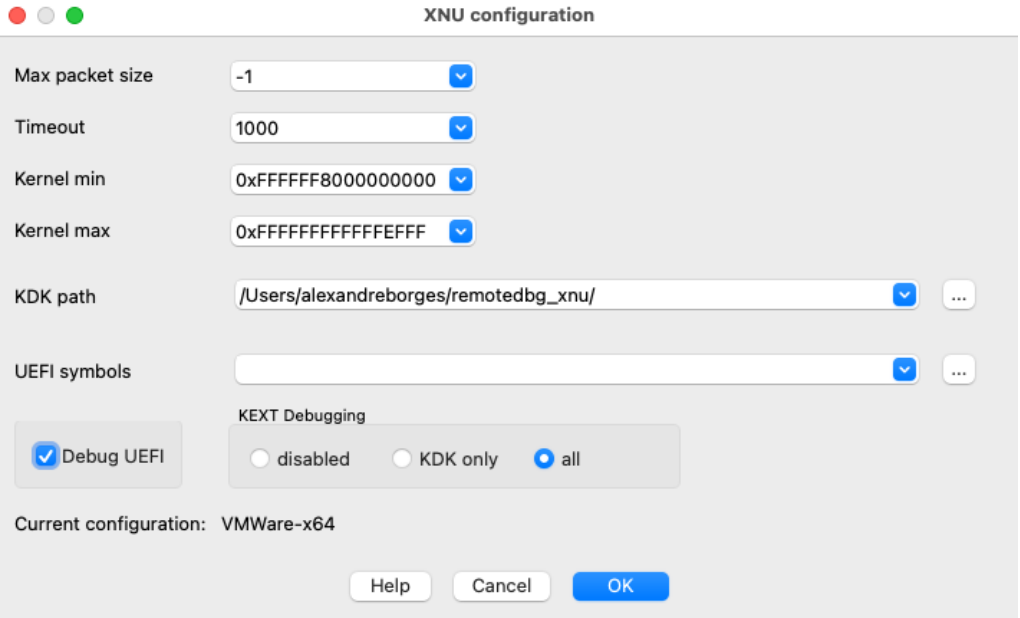


[Figure 38]: Kernel Extension segments

Thus, to improve our debugging experience again, copy three kernel collection files to our reserved copy of the KDK. As my host system (debugger) is exactly equal to the target, so the copy can be local, and whether the target’s version was different, so I should perform remote copy from target through SSH:

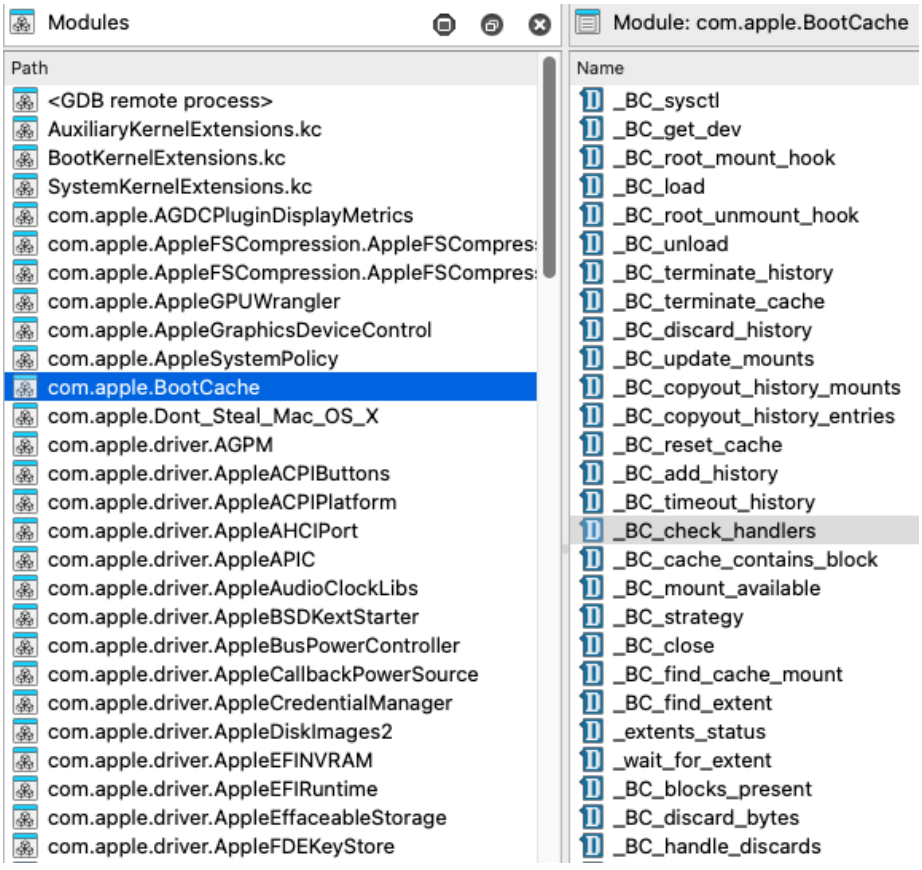
- **cd /Users/alexandreborges/remotedbg_xnu/KDK_14.6.1_23G93.kdk**
- **cp /System/Library/KernelCollections/BootKernelExtensions.kc .**
- **cp /System/Library/KernelCollections/SystemKernelExtensions.kc .**
- **cp /Library/KernelCollections/AuxiliaryKernelExtensions.kc** .

Afterwards, change the XNU configuration to the correct KDK path (I have specified one level up when compared to the previous setting) and also marked KEXT Debugging option as “all”:



[Figure 39]: Kernel Extension segments

Attach the debugger to the remote process again and you will see an extensive list of modules being debugged:



[Figure 40]: Extensive list of modules on debugger

07. Foundations

7.1. General Concepts

This section is only a review of the most important mechanisms to help readers to understand the next topics that will be treated in this article and the following ones. The XNU source code is always well-commented and useful for getting further information about functionalities and macOS/iOS concepts and you can retrieve the current version by running **git clone https://github.com/apple-oss-distributions/xnu**. Apple offers a clear explanation about the source code structure and its respective folders:

- **config:** it contains configurations for exported APIs for supported architecture and platform
- **SETUP:** it holds a set of tools used for configuring the kernel, versioning and kext symbol management.
- **EXTERNAL_HEADERS:** it contains headers sourced from other projects to avoid dependency cycles when building. These headers should be regularly synchronized when the source is updated.
- **libkern:** it contains the **C++ IOKit library code** for handling drivers and kexts.
- **libsa:** it contains kernel bootstrap code for startup
- **libsyscall:** it holds **syscall library interface** for user space programs
- **libkdd:** it contains source for **user library for parsing kernel data** like kernel chunked data.
- **makedefs:** it holds top level rules and **defines for kernel build**.
- **osfmk:** it contains **Mach** kernel-based subsystems
- **pexpert:** it holds platform specific code like **interrupt handling, atomics** etc.
- **security:** it contains **Mandatory Access Check** policy interfaces and related implementation.
- **bsd:** it contains **BSD subsystems code**
- **tools:** it contains a set of utilities for **testing, debugging, and profiling kernel**.

An absolute over-simplification (imprecise) is that the kernel is built by a BSD layer (high-layer) and Mach, which comes from NeXTStep (low-layer). In terms of the complete macOS, it is composed (from the higher layer to a lower layer) of an application layer, multiple media frameworks (packaged shared libraries with related resources), core services and operating system (including multiple runtime environments and external libraries such as libpcap, libpam, libxml, etc), and the mentioned kernel and its respective device drivers. Additionally, the macOS contains multiple property lists (composed of multiple key + value entries), which can be presented in XML, JSON, binary format, and there is a bundle application named **plutil** to validate and even convert its content between formats.

Eventually we will see a few pieces of Assembly code along with source code, but there is nothing really special. If readers do not remember about ARM Assembly, one of available options is to check my introductory article on macOS (MAS 08), which contains a limited section with the main information on the subject. At the same time, macOS presents Unix syscalls (most of them are documented, but not all, unfortunately), **Mach traps** (a kind of low-level syscalls), **machdep syscalls** and **diagnostic syscalls**. Furthermore, readers need to consider the fact there is support for Intel and ARM on XNU source code.

To illustrate the previous paragraph, a quick check in the content of **osfmk\mach\i386\syscall_sw.h** file shows the following:

```
61: #include <architecture/i386/asm_help.h>
62:
63: /*
64:  * Software interrupt codes for 32-bit system call entry:
65:  */
66: #define UNIX_INT     0x80
67: #define MACH_INT     0x81
68: #define MACHDEP_INT  0x82
69: #define DIAG_INT     0x83
70:
71: #if defined(__i386__)
72:
73: #ifndef KERNEL
74: /*
75:  * Syscall entry macros for use in libc:
76:  */
77: #define UNIX_SYSCALL_TRAP   \
78:     int $(UNIX_INT)
79: #define MACHDEP_SYSCALL_TRAP    \
80:     int $(MACHDEP_INT)
```

[Figure 41]: osfmk\mach\i386\syscall_sw.h file

We can see the syscall types mentioned previously in the figure above (lines 66-69). In the same file, a few lines ahead, all syscall classes are defined and macros used to build the system call numbers are shown:

```
123: /*
124:  * Syscall classes for 64-bit system call entry.
125:  * For 64-bit users, the 32-bit syscall number is partitioned
126:  * with the high-order bits representing the class and low-order
127:  * bits being the syscall number within that class.
128:  * The high-order 32-bits of the 64-bit syscall number are unused.
129:  * All system classes enter the kernel via the syscall instruction.
130:  *
131:  * These are not #ifdef'd for x86-64 because they might be used for
132:  * 32-bit someday and so the 64-bit comm page in a 32-bit kernel
133:  * can use them.
134:  */
135: #define SYSCALL_CLASS_SHIFT 24
136: #define SYSCALL_CLASS_MASK  (0xFF << SYSCALL_CLASS_SHIFT)
137: #define SYSCALL_NUMBER_MASK (~SYSCALL_CLASS_MASK)
138:
139: #define I386_SYSCALL_CLASS_MASK     SYSCALL_CLASS_MASK
140: #define I386_SYSCALL_ARG_BYTES_SHIFT    (16)
141: #define I386_SYSCALL_ARG_DWORDS_SHIFT   (I386_SYSCALL_ARG_BYTES_SHIFT + 2)
142: #define I386_SYSCALL_ARG_BYTES_NUM  (64) /* Must be <= sizeof(uu_arg) */
143: #define I386_SYSCALL_ARG_DWORDS_MASK    ((I386_SYSCALL_ARG_BYTES_NUM >> 2) -1)
144: #define I386_SYSCALL_ARG_BYTES_MASK (((I386_SYSCALL_ARG_BYTES_NUM -1)&~0x3) << I386_SYSCALL_ARG_BYTES_SHIFT)
145: #define I386_SYSCALL_NUMBER_MASK    (0xFFFF)
146:
147: #define SYSCALL_CLASS_NONE  0   /* Invalid */
148: #define SYSCALL_CLASS_MACH  1   /* Mach */
149: #define SYSCALL_CLASS_UNIX  2   /* Unix/BSD */
150: #define SYSCALL_CLASS_MDEP  3   /* Machine-dependent */
151: #define SYSCALL_CLASS_DIAG  4   /* Diagnostics */
152: #define SYSCALL_CLASS_IPC   5   /* Mach IPC */
153:
154: /* Macros to simpllfy constructing syscall numbers. */
155: #define SYSCALL_CONSTRUCT_MACH(syscall_number) \
156:             ((SYSCALL_CLASS_MACH << SYSCALL_CLASS_SHIFT) | \
157:              (SYSCALL_NUMBER_MASK & (syscall_number)))
158: #define SYSCALL_CONSTRUCT_UNIX(syscall_number) \
159:             ((SYSCALL_CLASS_UNIX << SYSCALL_CLASS_SHIFT) | \
160:              (SYSCALL_NUMBER_MASK & (syscall_number)))
161: #define SYSCALL_CONSTRUCT_MDEP(syscall_number) \
162:             ((SYSCALL_CLASS_MDEP << SYSCALL_CLASS_SHIFT) | \
163:              (SYSCALL_NUMBER_MASK & (syscall_number)))
164: #define SYSCALL_CONSTRUCT_DIAG(syscall_number) \
165:             ((SYSCALL_CLASS_DIAG << SYSCALL_CLASS_SHIFT) | \
166:              (SYSCALL_NUMBER_MASK & (syscall_number)))
```

[Figure 42]: osfmk\mach\i386\syscall_sw.h file (second part)

Those distinct types of syscalls, Unix syscalls and Mach traps are the most important ones.

Syscall are the most important channel and interface to the XNU, but they are not the only one because **I/O Kit** provides an aside channel to communication with device drivers, which are registered on macOS and can be listed through **IORegistry** as shown below:

```
alexandreborges@Mac ~ % ioreg -l | grep "<class" | grep " registered" | head -15
  +-o VMware20,1  <class IOPlatformExpertDevice, id 0x100000113, registered, matched, active, busy 0 (40319 ms), retain 27>
    +-o AppleACPIPlatformExpert  <class AppleACPIPlatformExpert, id 0x100000114, registered, matched, active, busy 0 (40005
ms), retain 32>
    | +-o IOSystemStateNotification  <class IOSystemStateNotification, id 0x100000118, registered, matched, active, busy 0 (
17 ms), retain 6>
    | +-o IOPMrootDomain  <class IOPMrootDomain, id 0x100000119, registered, matched, active, busy 0 (38 ms), retain 107>
    | +-o IOPCIMessagedInterruptController  <class IOPCIMessagedInterruptController, id 0x10000011c, registered, matched, ac
tive, busy 0 (9 ms), retain 7>
    | +-o cpus  <class IOPlatformDevice, id 0x10000011d, registered, matched, active, busy 0 (2 ms), retain 10>
    | +-o CP00@0  <class IOACPIPlatformDevice, id 0x10000011e, registered, matched, active, busy 0 (3976 ms), retain 8>
    | | +-o AppleACPICPU  <class AppleACPICPU, id 0x100000123, registered, matched, active, busy 0 (3954 ms), retain 8>
    | |   +-o AppleACPICPUInterruptController  <class AppleACPICPUInterruptController, id 0x100000139, registered, matched,
active, busy 0 (1 ms), retain 6>
    | |   +-o X86PlatformPlugin  <class X86PlatformPlugin, id 0x10000042f, registered, matched, active, busy 0 (3892 ms), re
tain 12>
    | |     +-o IOPlatformEnabler  <class IOPlatformPluginDevice, id 0x100000495, registered, matched, active, busy 0 (10 ms
), retain 6>
    | |     +-o AGPMEnabler  <class IOPlatformPluginDevice, id 0x100000496, registered, matched, active, busy 0 (10 ms), ret
ain 7>
    | +-o CP01@1  <class IOACPIPlatformDevice, id 0x10000011f, registered, matched, active, busy 0 (38 ms), retain 8>
    | | +-o AppleACPICPU  <class AppleACPICPU, id 0x100000124, registered, matched, active, busy 0 (0 ms), retain 6>
    | +-o CP02@2  <class IOACPIPlatformDevice, id 0x100000120, registered, matched, active, busy 0 (42 ms), retain 8>
alexandreborges@Mac ~ %
```

[Figure 43]: Listing registered drivers through IORegistry (truncated)

As readers already know, the output above is much longer than it, and for any listed drivers, it is possible to retrieve a list of its respective providers and classes clients, as shown below:

```
alexandreborges@Mac ~ % ioreg -f -r -t -n AppleKeyStore
+-o Root  <class IORegistryEntry, id 0x100000100, retain 8>
  +-o VMware20,1  <class IOPlatformExpertDevice, id 0x100000113, registered, matched, active, busy 0 (40588 ms), retain 27>
    +-o IOResources  <class IOResources, id 0x100000116, registered, matched, active, busy 0 (2023 ms), retain 60>
      +-o AppleKeyStore  <class AppleKeyStore, id 0x10000012b, registered, matched, active, busy 0 (25 ms), retain 50>
        | {
        |   "IOProbeScore" = 0
        |   "CFBundleIdentifier" = "com.apple.driver.AppleKeyStore"
        |   "IOMatchCategory" = "AppleKeyStore"
        |   "IOClass" = "AppleKeyStore"
        |   "IOPlatformWakeAction" = 1000
        |   "IOProviderClass" = "IOResources"
        |   "CFBundleIdentifierKernel" = "com.apple.driver.AppleKeyStore"
        |   "IOMatchedAtBoot" = Yes
        |   "IOUserClientClass" = "AppleKeyStoreUserClient"
        |   "IOGeneralInterest" = "IOCommand is not serializable"
        |   "IOResourceMatch" = "IOKit"
        | }
        |
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000003ea, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000003eb, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000003ec, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000003fd, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x100000400, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x10000040e, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x10000042b, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x100000522, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x100000525, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x10000052e, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x100000570, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x10000057f, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x10000058b, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x10000058c, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005a2, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005b2, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005b5, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005b6, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005e7, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005e8, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005f5, !registered, !matched, active, busy 0, retain 6>
        +-o AppleKeyStoreUserClient  <class AppleKeyStoreUserClient, id 0x1000005f8, !registered, !matched, active, busy 0, retain 6>
```

[Figure 44]: Additional details on a given driver (truncated)

Another interesting component of macOS are the kernel extensions (**kexts**), which are similar to "software drivers" on Windows, and that provides us additional features and communication with the kernel. They are stored in the **/System/Library/Extensions folder**, and the loaded ones can be listed running the following command:

```
alexandreborges@Mac ~ % kmutil showloaded | head -20
No variant specified, falling back to release
Index Refs Address            Size       Wired      Name (Version) UUID <Linked Against>
    1  108 0                  0          0          com.apple.kpi.bsd (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    2   11 0                  0          0          com.apple.kpi.dsep (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    3  130 0                  0          0          com.apple.kpi.iokit (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    4    0 0                  0          0          com.apple.kpi.kasan (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    5    0 0                  0          0          com.apple.kpi.kcov (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    6  135 0                  0          0          com.apple.kpi.libkern (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    7  117 0                  0          0          com.apple.kpi.mach (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    8   95 0                  0          0          com.apple.kpi.private (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
    9   75 0                  0          0          com.apple.kpi.unsupported (23.6.0) B8A95C25-3789-30A2-AE3C-7A48E0B91426 <>
   10   12 0xffffff8002fec000 0x7dff0    0x7dff0    com.apple.kec.corecrypto (14.0) 32E201B4-B19A-33DD-908F-850ED2F58E46 <9 8 7 6 3 1>
   11    0 0xffffff80018d7000 0x3000     0x3000     com.apple.kec.AppleEncryptedArchive (1) 16312272-301C-3543-B2B0-E3CFC5E5EF04 <10 8 7 6>
   12    0 0xffffff8001e7f000 0x4000     0x4000     com.apple.kec.Compression (1.0) 02125950-E87A-33AF-BAFC-BCFFC7E4BBC6 <10 8 7 6>
   13    2 0xffffff8002d3d000 0xdfe0     0xdfe0     com.apple.kec.Libm (1) 80081411-91D3-37FF-8783-E2EFC08C17C7 <6>
   14    0 0xffffff8002dee000 0x3000     0x3000     com.apple.kec.XrtHostedXnu (1) 7C5A38BE-633F-3516-B536-64E391220C8F <8 6>
   15    0 0xffffff8003099000 0x7000     0x7000     com.apple.kec.pthread (1) 53F9CA15-DA8B-3385-B9B6-7E3A6843AA07 <9 8 7 6 3 1>
   16    1 0xffffff80030af000 0x2fff     0x2fff     com.apple.driver.watchdog (1) 2D007C64-EF78-3276-A014-C76E314452E2 <9 8 7 6 3 1>
   17   20 0xffffff80022f8000 0x2000     0x2000     com.apple.iokit.IOACPIFamily (1.4) 04DDFFBA-0499-3590-B6C5-C40851E28B4F <9 8 6 3>
   18   23 0xffffff8002768000 0x32000    0x32000    com.apple.iokit.IOPCIFamily (2.9) 250D1260-447B-3BF3-96B2-C784699ED806 <9 8 7 6 3>
   19    8 0xffffff8001b88000 0x1bffe    0x1bffe    com.apple.driver.AppleSMC (3.1.9) B6D333BE-CC32-33DE-99BD-5824A3392BE0 <18 17 16 9 8 7 6 3 1>
```

[Figure 45]: Listing loaded kexts

Just in case there is not any kernel extension in **/System/Library/Extension** directory then your system could be using kernelcache, which is an optimized and pre-linked version of the kernel, which are compressed (IMG4 format) and includes all necessary drivers and kernel extensions. Depending on the system, kernelcache can be found as **/System/Volumes/Preboot/<id>/System/Library/Caches/com.apple.kernelcaches/kernelcache**. Later I will return to the kernelcache topic.

Alternatively, readers could use **kextstat | head -20** to accomplish the same work. Nowadays, due to the fact that extensions are able to perform a series of low-level tasks in the kernel space, and they provide high privileges and can be used as rootkit, then the option to load such kexts has been blocked by default, To enable it again, readers should boot the macOS into Recovery Mode, set the Security Level reduced security, allow the loading of third-party kexts and reboot the macOS again. Of course, it is not recommended to use **kexts** due to security risks. As we can realize, most of artifacts and mechanisms are directly or indirectly related to filesystems, and Apple offers a good approach about them because the system partition is kept apart from the data partition, where the former one can be updated without interfering with user data. It is not necessary to lay such details out because most readers already know about it, and even those who are comfortable with Linux easily understand the mount output:

```
alexandreborges@Mac ~ % mount
/dev/disk1s4s1 on / (apfs, sealed, local, read-only, journaled)
devfs on /dev (devfs, local, nobrowse)
/dev/disk1s2 on /System/Volumes/Preboot (apfs, local, journaled, nobrowse)
/dev/disk1s6 on /System/Volumes/VM (apfs, local, noexec, journaled, noatime, nobrowse)
/dev/disk1s5 on /System/Volumes/Update (apfs, local, journaled, nobrowse)
/dev/disk1s1 on /System/Volumes/Data (apfs, local, journaled, nobrowse, root data)
map auto_home on /System/Volumes/Data/home (autofs, automounted, nobrowse)
.host:/VMware Shared Folders on /Volumes/VMware Shared Folders (vmhgfs)
/dev/disk3s1 on /Library/Developer/CoreSimulator/Volumes/iOS_21F79 (apfs, local, nodev,
nosuid, read-only, journaled, noowners, noatime, nobrowse)
```

[Figure 46]: Common mounted file systems (except the VMware Shared Folder)

There are some observations:

- **Root filesystem**: it is an Apple File System (APFS), sealed (cryptographically signed) and read-only, enforcing its anti-tampering characteristic. Actually, the SSV (Signed System Volume) feature, which is indicated by the sealed attribute, ensures that the system content is verified in runtime.
- **Preboot volume:** it is used, as expected, for the pre-boot task that is required for booting the operating system.
- **VM volume:** this is a special filesystem used for virtual memory and, no doubt, the no-exec attribute is a strong indication that binaries cannot be executed from it.
- **Data volume:** it contains application and user data, as expected.
- **Update volume:** it is basically used for system updates.

There are many further details that can be discussed about such volumes as well as there is a series of possible file system that could be created (check **/System/Library/Filesystems**), but they are needless right now. I am not forgetting that **disk images (.dmg)**, which are still used for software bundles, mount its content in ***/Volumes/<label>*** through either **Finder** or even hdiutil command (**hdiutil attach <dmg_image>**). It is really a pretty uncomplicated way to install applications because such a technique copies files to a filesystem, which macOS does this task straightful on purpose being enough to create the due file structure under **/Application/<application_name>.app**. Furthermore, built-in Apple applications are secure against tampering or even unauthorized removal because of the SIP protection (it will be commented later). **Dmg files** are not the only format used for application installation, and pkg files are also a format used for packaging programs. In the iOS context, customers download applications from Apple store, and downloaded applications (**.ipa files**) are digitally signed, encrypted (FairPlay) and even can be installed into **/Application** directory (dedicated to system applications) and are installed under **/var/containers/Bundle/Application** directory (after having its respective Payload folder renamed).

Back to macOS, it keeps a central database of installed applications that is named **LaunchServices database**, and curiously there is a strangely hidden tool named **lsregister** in **/System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/LaunchServices.framework/Versions/A/Support** folder that can be used to dump the mentioned database. However, as the output is insanely long, a brief filtered list follows below:

```
alexandreborges@Mac Support % ./lsregister -dump | grep "bundle id:" | head -15
bundle id:                    LockScreen (0x1bd0)
bundle id:                    Share Screen Request (0x1bd4)
bundle id:                    SSDragHelper (0x1bd8)
bundle id:                    Remote Desktop Message (0x1bdc)
bundle id:                    MakePDF (0x216c)
bundle id:                    SSInvitationAgent (0x1be0)
bundle id:                    LinkedNotesUIService (0x2170)
bundle id:                    SSAssistanceCursor (0x1be4)
bundle id:                    group.is.workflow.my.app (0x1658)
bundle id:                    Computer (0x2178)
bundle id:                    SSMenuAgent (0x1be8)
bundle id:                    com.apple.ctcategories.service (0x165c)
bundle id:                    RegisterPluginIM (0x1bec)
bundle id:                    Hopper Disassembler v4 (0x1660)
bundle id:                    ShortcutsActions (0x1bf0)
```

[Figure 47]: LaunchService database dump (filtered and truncated)

The **LaunchServices framework** is controlled and managed by the **lsd daemon**, which enforces a list of entitlements (rights and privileges to do something), and it is responsible for handling static registration for such services, while the **launchservicesd (LaunchDaemon)** is responsible for the dynamic runtime related to such services. As readers could notice, applications and services depend on different and a multitude of runtime dependencies, which is solved by the dynamic linker (dyld). I've already written about Mach-O and dyld in part 8 of **Malware Analysis Series (MAS)**, and I do not want repeat myself again here even being a different context (https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/). Eventually a few topics associated directly and indirectly with dyld could be mentioned such as code injection using **DYLD_INSERT_LIBRARIES, method swizzling** and **interposing**.

DYLD_INSERT_LIBRARIES code injection has a terribly similar counterpart on Linux, and it allows dynamically loaders libraries before those ones specified by the application, which can be done using a syntax like this one on the command line:

- **DYLD_INSERT_LIBRARIES=somelibrary.dylib ./application**

Nowadays there are protections such as code signing enforcement, notarization, different forms of library validation prevent technique to work as expected, and PAC (Pointer Authentication Code), which will be commented later, also make any exploitation using this technique harder than usual.

Method swizzling is a programming technique in Objective-C for changing the selector and, as consequence, also altering the method implementation being invoked in runtime. Actually, the method implementation is not changed, but another implementation for such a method is called in place by changing the selector (selector A → method A → implementation A, selector B → method B → implementation B). This exploitation technique is focused on Objective-C runtime, so it targets Objective-C methods.

Method interposing also offers the replacement of functions with distinguished implementations, but it is not focused on Objective-C and works by including a new section into the Mach-O binary which contains the new address of a target function. Additionally, the interposing technique provides us with a bonus because it does not act if the call comes from the own binary being interposed, the call goes to the original function. The entire interposing effect is got using **DYLD_INTERPOSE macro**, which makes it a technique that can only be used in loading time. The **dynamic interposing** can be deployed using **dyld_dynamic_interpose( )** function, which makes part of the dynamic linker (dyld).

7.2. Mach

So far, we have always viewed macOS from a high-level perspective, but as I have mentioned previously, macOS is a composition of BSD, Mach, and I/O Kit driver framework in general. There are many important directories and files in the XNU structure, and a few of them that are worth to pay attention are:

- **bsd/kern**
- **iokit/IOKit**
- **iokit/Kernel/**
- **libkern/libkern**
- **osfmk/UserNotification**

- **osfmk/device**
- **osfmk/ipc**
- **osfmk/kern**
- **osfmk/vm**

Mach is responsible for the low-level communication between different kernel components and also high layers interfaces, so it provides the necessary abstraction to BSD layer in several aspects such as tasks, threads, ports, messages and mainly memory objects. These mentioned structures (types) that are not the only involved with Mach, and there are other ones like processor, host, alarm and so on (check **osfmk\mach\mach_types.h**). Additionally, the BSD layer and Mach layer are bound in many ways.

To avoid getting into unnecessary details at this moment (I will return to details later), the usual process that are well-known from other operating systems are represented in Mach as a dual-component that is a task, which does not execute, and a thread that is the true running object that executes inside of the task, and it shares similarities with Linux and Windows by owning its own kernel stack, exception handers, processor registers and so on. Nonetheless, there is not a parent-child relationship between tasks. Task definition can be found in **osfmk\kern\task.h** file. As readers might predict, there is an internal routine named **task_create_internal**, as shown below (**osfmk\kern\task_create_internal**):

```
1475: /* Create a task, but leave the task ports disabled */
1476: kern_return_t
1477: task_create_internal(
1478:     task_t              parent_task,            /* Null-able */
1479:     proc_ro_t           proc_ro,
1480:     coalition_t         *parent_coalitions __unused,
1481:     boolean_t           inherit_memory,
1482:     boolean_t           is_64bit,
1483:     boolean_t           is_64bit_data,
1484:     uint32_t            t_flags,
1485:     uint32_t            t_flags_ro,
1486:     uint32_t            t_procflags,
1487:     uint8_t             t_returnwaitflags,
1488:     task_t              child_task)
1489: {
1490:     task_t                  new_task;
1491:     vm_shared_region_t      shared_region;
1492:     ledger_t                ledger = NULL;
1493:     struct task_ro_data     task_ro_data = {};
1494:     uint32_t                parent_t_flags_ro = 0;
1495:
1496:     new_task = child_task;
1497:
1498:     if (task_ref_count_init(new_task) != KERN_SUCCESS) {
1499:         return KERN_RESOURCE_SHORTAGE;
1500:     }
1501:
1502:     /* allocate with active entries */
1503:     assert(task_ledger_template != NULL);
1504:     ledger = ledger_instantiate(task_ledger_template, LEDGER_CREATE_ACTIVE_ENTRIES);
1505:     if (ledger == NULL) {
1506:         task_ref_count_fini(new_task);
1507:         return KERN_RESOURCE_SHORTAGE;
1508:     }
```

[Figure 48]: Task creation (truncated)

As a side note, although we have referred to Mach aspects up to this point, the BSD process representation is like other systems such Linux, for example. As we can notice, there are relevant differences between Mach tasks and BSD processes because the Mach task structure contains information IPC and addresses, while BSD process structure contains usual information such as credentials and file descriptors.

The visualization of **task structure** and **proc structure** makes such difference clearer:

```
struct task {
    /* Synchronization/destruction information */
    decl_lck_mtx_data(, lock); /* Task's lock */
    os_refcnt_t     ref_count;      /* Number of references to me */

#if DEVELOPMENT || DEBUG
    struct os_refgrp *ref_group;
    lck_spin_t        ref_group_lock;
#endif /* DEVELOPMENT || DEBUG */

    bool            active;         /* Task has not been terminated */
    bool            ipc_active;     /* IPC with the task ports is allowed */
    bool            halting;        /* Task is being halted */
    bool            message_app_suspended;  /* Let iokit know when pidsuspended */

    /* Virtual timers */
    uint32_t        vtimers;
    uint32_t loadTag; /* dext ID used for logging identity */

    /* Globally uniqueid to identify tasks and corpses */
    uint64_t        task_uniqueid;

    /* Miscellaneous */
    vm_map_t        XNU_PTRAUTH_SIGNED_PTR("task.map") map; /* Address space description */
    queue_chain_t   tasks;  /* global list of tasks */
    struct task_watchports *watchports; /* watchports passed in spawn */
    turnstile_inheritor_t returnwait_inheritor; /* inheritor for task_wait */

    /* Threads in this task */
    queue_head_t            threads;
    struct restartable_ranges *t_rr_ranges;
```

[Figure 49]: task structure (truncated)

The BSD process structure (**struct proc**) is also shown below:

```
struct proc {
    union {
        LIST_ENTRY(proc) p_list;                /* List of all processes. */
        struct smr_node  p_smr_node;
    };
    struct  proc *  XNU_PTRAUTH_SIGNED_PTR("proc.p_pptr") p_pptr;   /* Pointer to parent process.(LL) */
    proc_ro_t       p_proc_ro;
    pid_t           p_ppid;                 /* process's parent pid number */
    pid_t           p_original_ppid;        /* process's original parent pid number, doesn't change if reparented */
    pid_t           p_pgrpid;               /* process group id of the process (LL)*/
    uid_t           p_uid;
    gid_t           p_gid;
    uid_t           p_ruid;
    gid_t           p_rgid;
    uid_t           p_svuid;
    gid_t           p_svgid;
    pid_t           p_sessionid;
    uint64_t        p_puniqueid;            /* parent's unique ID - set on fork/spawn, doesn't change if reparented. */

    lck_mtx_t       p_mlock;                /* mutex lock for proc */
    pid_t           p_pid;                  /* Process identifier for proc_find. (static)*/
    char            p_stat;                 /* S* process status. (PL)*/
    char            p_shutdownstate;
    char            p_kdebug;               /* P_KDEBUG eq (CC)*/
    char            p_btrace;               /* P_BTRACE eq (CC)*/

    LIST_ENTRY(proc) p_pglist;              /* List of processes in pgrp (PGL) */
    LIST_ENTRY(proc) p_sibling;             /* List of sibling processes (LL)*/
    LIST_HEAD(, proc) p_children;           /* Pointer to list of children (LL)*/
    TAILQ_HEAD(, uthread) p_uthlist;        /* List of uthreads (PL) */

    struct smrq_slink p_hash;               /* Hash chain (LL)*/
```

[Figure 50]: BSD proc structure (truncated)

Mach ports, which are a kind of object references (it is an imprecise definition, but it reasonable for now), work as an **IPC communication channel** that is implemented as a message key and has rights (capabilities) associated with them and that can determine whether a task would be allowed or not to send a message

to the port or even which task could receive messages that are addressed to them. Therefore, a task has to hold the necessary rights to manipulate a given port. As an extension of the rules, multiple tasks can contain "**send rights**" to a target port, even though only one task can have "**receive rights**" to the same port. There are different port rights, and they can be checked in **osfmk\mach\port.h** file:

```
#if XNU_KERNEL_PRIVATE
__enum_closed_decl(mach_port_right_t, uint32_t, {
    MACH_PORT_RIGHT_SEND            = 0,
    MACH_PORT_RIGHT_RECEIVE         = 1,
    MACH_PORT_RIGHT_SEND_ONCE       = 2,
    MACH_PORT_RIGHT_PORT_SET        = 3,
    MACH_PORT_RIGHT_DEAD_NAME       = 4,
    MACH_PORT_RIGHT_LABELH          = 5, /* obsolete right */
    MACH_PORT_RIGHT_NUMBER          = 6, /* right not implemented */
});

#define MACH_PORT_RIGHT_VALID_TRANSLATE(right) \
    ((right) >= MACH_PORT_RIGHT_SEND && (right) <= MACH_PORT_RIGHT_DEAD_NAME)
#else
typedef natural_t mach_port_right_t;

#define MACH_PORT_RIGHT_SEND            ((mach_port_right_t) 0)
#define MACH_PORT_RIGHT_RECEIVE         ((mach_port_right_t) 1)
#define MACH_PORT_RIGHT_SEND_ONCE       ((mach_port_right_t) 2)
#define MACH_PORT_RIGHT_PORT_SET        ((mach_port_right_t) 3)
#define MACH_PORT_RIGHT_DEAD_NAME       ((mach_port_right_t) 4)
#define MACH_PORT_RIGHT_LABELH          ((mach_port_right_t) 5) /* obsolete right */
#define MACH_PORT_RIGHT_NUMBER          ((mach_port_right_t) 6) /* right not implemented */
#endif
```

[Figure 51]: Mach port rights

As such ports have an interface and communication behavior (by the way, a **unidirectional channel**), they are used as **protected access to different kernel and system resources such as the mentioned ports**, **threads and even memory**, which will be discussed later, but also show that all of these resources have a respective port assigned. For example, a **Mach task** has a kernel port and exception ports associated with it. The protected access characteristic of a port comes from the need of having required rights (capability) to send or receive any message to the port. For sure, all **Mach threads** (also known as kernel thread) from a task have natural access to ports of the respective task, but a task (and it threads) cannot access ports from other threads unless it has the appropriate rights. In terms of referential information, the name of a Mach port is represented by an integer in a resembling way of Linux file descriptors. As we are talking about messages, **Mach IPC messages** are used by threads to communicate with each other and also provide quite an interesting feature of being able to asynchronously transmit port rights (capabilities) between tasks. No doubt, it is a bit strange this concept of sending or receiving rights to/from a port, but readers will get accustomed to it and will see further details later.

Following the same subject, macOS also provides us with the **Mach traps** concept, which are terribly similar to system calls from other systems, but they do not allow direct access to Mach kernel services through such traps. The point of communication offered by Mach traps is for user programs to access server counterparts, and these provide some of requested kernel services. In other words, it would be something like **client ←→ server ←→ kernel services**. Furthermore, **RPC (Remote Procedure Call)** is the foundation of Mach communication, and associated Mach services normally make use of **MIG (Mach Interface Generator), which provides an excellent framework and simplify the creation of associated Mach clients and servers by generating all the boilerplates**. As a suggestion, check **osfmk\mach\mach_traps.h** file for distinct types of Mach traps.

Based on the exposed structures and objects, **Mach ports** are initially quite an unusual concept because:

- They have and offer a kind of **access interface and communication object**.
- They can be accessed only with the requested port rights (send, send_once, receive and so on).
- **Port rights** can be passed through messages.
- **Ports** are usually **unidirectional**.
- A port is simply a **kernel object**.
- The **port name** is a handle with limited reach and restricted to the task's scope.

Other port types are important such as:

- **task port:** it offers an IPC channel for communication between tasks, where each task has a respective **task port** that allows the interaction with the kernel and other tasks.

- **thread port:** it permits IPC communication between threads and kernel by sending and receiving message. Additionally, there are **two types of thread ports**:
  - **thread control port**: it can be used to **manage and control** a thread by executing operations including suspending and resuming a thread.
  - **exception port:** it is used to manage thread exceptions and errors.

- **Host port:** it offers access to the **I/O Register** and **processor set**, being that it also manages resources such as memory allocation and power management.

- **host_priv port**: it provides access to **sensitive and privileged system information**, which are accessible from root or system processes with elevated privileges.

- **host special port:** they are **dedicated for system usage** and not to be used by user applications, are most of the time **associated with a specific service (daemon) for system-level communication** and services (security services, Apple services, System Ports).

- **device ports**: they are **created and used for communication with hardware devices** when a driver needs to retrieve information and send commands.

- **memory object ports:** they are used to **manage virtual memory objects**, and in special they are used for **implementing shared regions between tasks and paging**.

- **security server port:** it is used for **communication with macOS components** that manage authentication and authorization

- **notification ports:** they are used **to send and receive system notification and alerts** in a scenario that a **process can register for notification by creating exactly a notification port**, **and associating the types of events that it is willing to receive**. Besides process monitoring, there are also events related to hardware and applications.

- **IPC ports:** they are general purpose ports used for communication between processes, and there are two classes of IPC ports: **port rights** (capabilities) and **special ports** (described above)

- **I/O Kit ports:** they are used for **controlling and monitoring I/O Kit drivers and devices**.

There are **special ports** that are provided by the kernel, and other ones not:

```
 74: /*
 75:  * Always provided by kernel (cannot be set from user-space).
 76:  */
 77: #define HOST_PORT                       1
 78: #define HOST_PRIV_PORT                  2
 79: #define HOST_IO_MAIN_PORT               3
 80: #define HOST_MAX_SPECIAL_KERNEL_PORT    7 /* room to grow */
 81:
 82: #define HOST_LAST_SPECIAL_KERNEL_PORT   HOST_IO_MAIN_PORT
 83:
 84: /*
 85:  * Not provided by kernel
 86:  */
 87: #define HOST_DYNAMIC_PAGER_PORT         (1 + HOST_MAX_SPECIAL_KERNEL_PORT)
 88: #define HOST_AUDIT_CONTROL_PORT         (2 + HOST_MAX_SPECIAL_KERNEL_PORT)
 89: #define HOST_USER_NOTIFICATION_PORT     (3 + HOST_MAX_SPECIAL_KERNEL_PORT)
 90: #define HOST_AUTOMOUNTD_PORT            (4 + HOST_MAX_SPECIAL_KERNEL_PORT)
 91: #define HOST_LOCKD_PORT                 (5 + HOST_MAX_SPECIAL_KERNEL_PORT)
 92: #define HOST_KTRACE_BACKGROUND_PORT     (6 + HOST_MAX_SPECIAL_KERNEL_PORT)
 93: #define HOST_SEATBELT_PORT              (7 + HOST_MAX_SPECIAL_KERNEL_PORT)
 94: #define HOST_KEXTD_PORT                 (8 + HOST_MAX_SPECIAL_KERNEL_PORT)
 95: #define HOST_LAUNCHCTL_PORT             (9 + HOST_MAX_SPECIAL_KERNEL_PORT)
 96: #define HOST_UNFREED_PORT               (10 + HOST_MAX_SPECIAL_KERNEL_PORT)
 97: #define HOST_AMFID_PORT                 (11 + HOST_MAX_SPECIAL_KERNEL_PORT)
 98: #define HOST_GSSD_PORT                  (12 + HOST_MAX_SPECIAL_KERNEL_PORT)
 99: #define HOST_TELEMETRY_PORT             (13 + HOST_MAX_SPECIAL_KERNEL_PORT)
100: #define HOST_ATM_NOTIFICATION_PORT      (14 + HOST_MAX_SPECIAL_KERNEL_PORT)
101: #define HOST_COALITION_PORT             (15 + HOST_MAX_SPECIAL_KERNEL_PORT)
102: #define HOST_SYSDIAGNOSE_PORT           (16 + HOST_MAX_SPECIAL_KERNEL_PORT)
103: #define HOST_XPC_EXCEPTION_PORT         (17 + HOST_MAX_SPECIAL_KERNEL_PORT)
104: #define HOST_CONTAINERD_PORT            (18 + HOST_MAX_SPECIAL_KERNEL_PORT)
105: #define HOST_NODE_PORT                  (19 + HOST_MAX_SPECIAL_KERNEL_PORT)
106: #define HOST_RESOURCE_NOTIFY_PORT       (20 + HOST_MAX_SPECIAL_KERNEL_PORT)
107: #define HOST_CLOSURED_PORT              (21 + HOST_MAX_SPECIAL_KERNEL_PORT)
108: #define HOST_SYSPOLICYD_PORT            (22 + HOST_MAX_SPECIAL_KERNEL_PORT)
109: #define HOST_FILECOORDINATIOND_PORT     (23 + HOST_MAX_SPECIAL_KERNEL_PORT)
110: #define HOST_FAIRPLAYD_PORT             (24 + HOST_MAX_SPECIAL_KERNEL_PORT)
111: #define HOST_IOCOMPRESSIONSTATS_PORT    (25 + HOST_MAX_SPECIAL_KERNEL_PORT)
112: #define HOST_MEMORY_ERROR_PORT          (26 + HOST_MAX_SPECIAL_KERNEL_PORT)
113: #define HOST_MANAGEDAPPDISTD_PORT       (27 + HOST_MAX_SPECIAL_KERNEL_PORT)
114: #define HOST_DOUBLEAGENTD_PORT          (28 + HOST_MAX_SPECIAL_KERNEL_PORT)
115:
116: #define HOST_MAX_SPECIAL_PORT           HOST_DOUBLEAGENTD_PORT
117: /* MAX = last since rdar://59872249 */
```

[Figure 52]: Special ports

Additionally, there are many contrasting functions to work with ports, and a few are one of them:

- **mach_port_names**
- **mach_port_type**
- **mach_port_rename**
- **mach_port_allocate_name**
- **mach_port_allocate**
- **mach_port_destroy**
- **mach_port_get_refs**
- **mach_port_peak**
- **mach_port_extract_right**
- **mach_port_kernel_object**

A recommended file to read a good description on each function above is **osfmk\mach\mach_port.defs** file, but we will see some of these functions later. We can retrieve the Mach port list from a specific process (the font is small because there are too many columns, unfortunately):

- **ps aux | grep “login -pf”**
- **lsmp -p 9332 | head -25**

```
Mac:~ root# ps aux | grep "login -pf"
root              9332   0.0  0.1 34149912   3928 s002  Ss    9:55PM   0:00.02 login -pf alexandreborges
root              2038   0.0  0.1 34149912   3336 s001  Ss   29Oct24   0:00.03 login -pf alexandreborges
root              1087   0.0  0.1 34149912   2716 s000  Ss   29Oct24   0:00.13 login -pf alexandreborges
root              9361   0.0  0.0 34121296    664 s002  S+    9:58PM   0:00.00 grep login -pf
Mac:~ root#
Mac:~ root# lsmp -p 9332 | head -25
Process (9332) : login
  name      ipc-object    rights     flags   boost  reqs  recv  send sonce oref  qlimit  msgcount  context            identifier  type
---------   ----------  ----------  --------  -----  ----  ----  ----  ----  ----  ------  --------  ------------------ ----------  ------------
0x00000103  0x11274cab  send        --------        ---           1                                                      0x00000000  THREAD-CONTROL (0x59034)
0x00000203  0x1136fcbb  send        --------        ---           2                                                      0x00000000  TASK-CONTROL SELF (9332) login
0x00000303  0x1135ef1b  recv        --------     0  ---      1               N        5         0  0x0000000000000000
0x00000403  0x1130158b  recv        --------     0  ---      1               N        5         0  0x0000000000000000
0x00000503  0x1136d3c3  recv        --------     0  ---      1               N        5         0  0x0000000000000000
0x00000607  0x0f9efdeb  send        --------        ---           2                                                      0x00000000  HOST-PRIV
0x00000703  0x1137e203  recv        --------     0  ---      1               N        5         0  0x0000000000000000
0x00000803  0x11369593  recv        --------     0  ---      1               N        5         0  0x0000000000000000
0x00000903  0x0f9f0933  send        --------        ---           1                                                      0x00000000  CLOCK
0x00000a03  0x11387e7b  send        --------        ---           1                                                      0x00000000  SEMAPHORE
0x00000b03  0x0f9f05a3  send        --------        ---           1                                                      0x00000000  HOST
0x00000c03  0x0f9d0033  send        --------        ---           1                                                      0x00000000  VOUCHER
0x00000d03  0x0f9c99bb  send        --------        ---           1         ->       32         0  0x0000000000000000 0x00001b03  (150) notifyd
0x00000e03  0x0f9c432b  send        --------        ---           1         ->       16         0  0x0000000000000000 0x00005707  (127) opendirectoryd
0x00000f03  0x11318213  send        --------        ---           1         ->        6         0  0x0000000000000000 0x0005063f  (127) opendirectoryd
0x00001003  0x11313633  recv,send  ---GS---     0  ---      1     1          Y        5         0  0x0000000000000000
                +       send        --------        ---           1         <-                                        0x00066257  (86) logd
0x00001107  0x0f9c70e3  send        --------        ---           1         ->       16         0  0x0000000000000000 0x00003703  (127) opendirectoryd
0x00001203  0x1137eaeb  send        --------        ---           1         ->        1         0  0x0000000000000000 0x000631d3  (86) logd
0x00001303  0x1130dae3  recv        ---GS---     0  ---      1               Y        6         0  0x0000000000000000
                +       send        --------        ---           1         <-                                        0x0004fa07  (127) opendirectoryd
0x00001407  0x1135d62b  recv        ---GSI--     0  ---      1           1   Y        6         0  0x0000000000000000
Mac:~ root#
```

[Figure 53: Mach ports list (truncated)

The plus signal (+) indicates a remote client connected to the given local port. More on Mach ports later.

7.3. Memory

Memory management is a fascinating topic, but I want to keep our scope reduced and only present basic ideas that likely readers already know about. Actually, we can touch many dissimilar parts of memory while describing definitions and involved structures: low-level (Mach level) and high-level heap (in special), and for exploitation it is the most attractive subject, by far.

MacOS/iOS also treats memory as 4K block (regular size), even though it might be larger than it (mainly in modern iOS devices that offer kernel pages with 16KB). Well-known concepts such as paging, copy-on-write, PFN (page frame number), anonymous memory, swapping and memory status such as free, active, and inactive (pages previously used, but not any longer) are the same, or in a few cases, almost the same. Eventually, names such as wired (resident) or even speculative (fetched in operations like read-ahead) might be initially strange, but there is nothing really new and about which you should worry.

There are many memory APIs that are also grouped in BSD system calls and Mach system calls, which are defined in a few files. Additionally, most of these functions are provided by files such as:

- **bsd\sys\malloc.h**
- **libsyscall\mach\mach_vm.c**
- **osfmk\vm\vm_map.h**
- **osfmk\vm\vm_map_internal.h**

From these files we can find prototypes and definitions of memory functions. The first ones (malloc.h) expose the first usual and very primitive BSD functions:

```
__BEGIN_DECLS
/* [ML] */
int     mlockall(int);
int     munlockall(void);
/* [MR] */
int     mlock(const void *, size_t);
#ifndef _MMAP
#define _MMAP
/* [MC3]*/
void *  mmap(void *, size_t, int, int, int, off_t) __DARWIN_ALIAS(mmap);
#endif
/* [MPR] */
int     mprotect(void *, size_t, int) __DARWIN_ALIAS(mprotect);
/* [MF|SIO] */
int     msync(void *, size_t, int) __DARWIN_ALIAS_C(msync);
/* [MR] */
int     munlock(const void *, size_t);
/* [MC3]*/
int     munmap(void *, size_t) __DARWIN_ALIAS(munmap);
/* [SHM] */
int     shm_open(const char *, int, ...);
int     shm_unlink(const char *);
/* [ADV] */
int     posix_madvise(void *, size_t, int);

#if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
int     madvise(void *, size_t, int);
int     mincore(const void *, size_t, char *);
int     minherit(void *, size_t, int);
#endif
```

[Figure 54]: malloc.h file (truncated)

A recommended file to check for constants, which supplement the previous one, is **bsd\sys\mman.h**:

```
/*
 * Protections are chosen from these bits, or-ed together
 */
#define PROT_NONE       0x00    /* [MC2] no permissions */
#define PROT_READ       0x01    /* [MC2] pages can be read */
#define PROT_WRITE      0x02    /* [MC2] pages can be written */
#define PROT_EXEC       0x04    /* [MC2] pages can be executed */

/*
 * Flags contain sharing type and options.
 * Sharing types; choose one.
 */
#define MAP_SHARED      0x0001          /* [MF|SHM] share changes */
#define MAP_PRIVATE     0x0002          /* [MF|SHM] changes are private */
#if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
#define MAP_COPY        MAP_PRIVATE     /* Obsolete */
#endif  /* (!_POSIX_C_SOURCE || _DARWIN_C_SOURCE) */

/*
 * Other flags
 */
#define MAP_FIXED        0x0010 /* [MF|SHM] interpret addr exactly */
#if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
#define MAP_RENAME       0x0020 /* Sun: rename private pages to file */
#define MAP_NORESERVE    0x0040 /* Sun: don't reserve needed swap area */
#define MAP_RESERVED0080 0x0080 /* previously unimplemented MAP_INHERIT */
#define MAP_NOEXTEND     0x0100 /* for MAP_FILE, don't change file size */
#define MAP_HASSEMAPHORE 0x0200 /* region may contain semaphores */
#define MAP_NOCACHE      0x0400 /* don't cache pages for this mapping */
#define MAP_JIT          0x0800 /* Allocate a region that will be used for JIT purposes */

/*
 * Mapping type
 */
#define MAP_FILE        0x0000  /* map from file (default) */
#define MAP_ANON        0x1000  /* allocated from memory, swap space */
#define MAP_ANONYMOUS   MAP_ANON
```

[Figure 55]: mman.h file (truncated)

Some of these BSD functions, which are well-known, follow:

- **mmap:** this function map files or even devices into memory. This function is defined in **libsyscall\wrappers\unix03\mmap.c** file.
- **munmap:** this function unmap files from memory.
- **mprotect:** this function changes the protection attributes of a given memory region, and some of these protections are PROT_NONE (0x00), PROT_READ (0x01), PROT_WRITE (0x02) and PROT_EXEC (0x04).
- **msync:** this function synchronizes a given mapped memory region with a file in the file system.
- **mlock:** this function locks a set of physical pages into memory (like resident memory) to prevent such pages being paging-out later.
- **madvise:** this function provides a kind of advice to the system (kernel, in special) about the intended use of the memory.
- **minherit:** this function controls the inheritance characteristics of memory pages such as VM_INHERIT_NONE (pages are not inherited by child address), VM_INHERIT_COPY (pages are copied to child processes) and VM_INHERIT_SHARE (pages are shared with child process).

In terms of Mach layer, based on other files such as **libsyscall\mach\mach_vm.c**, **osfmk\vm\vm_map.h** and **osfmk\vm\vm_map_internal.h**, some of Mach functions arise and are recommended to know:

- **mach_vm_allocate:** this function allocates a region of virtual memory within the assigned space for a specified task.
- **mach_vm_deallocate:** this function deallocates a region of virtual memory within the assigned space for a specified task.
- **mach_vm_protect:** this function changes memory protection attributes of a region of memory. Memory attributes can be **VM_PROT_READ, VM_PROT_WRITE** and **VM_PROT_EXECUTE**.
- **vm_allocate:** this function allocates a region of memory.
- **vm_deallocate:** this function deallocates a region of memory.
- **vm_protect:** this function changes a given memory page protection.
- **mach_vm_map:** this function maps a region of memory in a task.
- **mach_vm_remap:** this function changes the region of an existing mapping.
- **mach_vm_remap_new:** this function remaps a range of memory from one task to another.
- **mach_vm_read:** this function reads a range of virtual memory from a given task**.**
- **vm_map:** this function creates a memory mapping in a given task.
- **vm_remap:** this function maps a region of memory from one task to another or even within the same task.
- **vm_read:** this function reads a range of virtual memory from a given task.
- **vm_remap_new:** this function remaps a range of memory from one task to another.
- **vm_read:** this function reads a range of virtual memory from a given task**.**
- **mach_vm_purgable_control:** this function manages the purgability of a given memory region**.** By the way, purgable memory is a memory that can be freed when the system needs more memory.
- **vm_purgable_control:** this function manages the purgability of a given memory region**.**
- **vm_map_create:** this function creates an empty VM map, where the VM map offers a representation of an address space.
- **vm_map_deallocate:** this function deallocates a VM map.
- **vm_map_wire:** this function marks a region as non-pageable (locked in memory).

- **vm_map_unwire:** this function marks a wired region as pageable again.

There are a multitude of concepts, ideas and definitions related to memory in macOS, mainly when we involve low-level (Mach) details:

- A **task** can retrieve a **memory object port** from a memory manager.
- An address space of a task (at Mach level) is represented by a **VM map** (**struct _vm_map** in **osfmk\vm\vm_map_xnu.h** file), which is composed of a doubly-linked list of memory regions and even physical map structures.
- A **VM map** (**struct _vm_map** in **osfmk\vm\vm_map_xnu.h file)** is associated with a **VM object (struct vm_object** in **osfmk\vm\vm_object_xnu.h** file), and it is considered a set of **VM map entries** (**struct vm_map_entries** in **osfmk\vm\vm_map_entry** file).
- A **VM object** (**struct vm_object** in **osfmk\vm\vm_object_xnu.h** file) is represented by a memory object, which is associated with a **memory object port**, whose rights (capabilities) are applied. Therefore, the memory object is a kind of interface to a source of data.
- A **VM map entry** is composed by a list of **VM pages** (**vm_page** in **osfmk\vm\vm_page** file) that is mapped in a task.
- In other words, a **VM map** represents **an address space managed by the Mach VM**.
- Mach VM is the pivotal point for the whole configuration of regions.
- A **pager**, which is associated with a memory object port that is the real provider of data, is responsible for managing **memory objects** and **pages**.
- There are several types of pagers, and the **default pager** is responsible for moving data between memory and swap space.
- At end of a mapped memory used by a task, the respective range is **deallocated,** and all **references** are also given up.

From a user-mode (high-level) point of view, most of the time memory allocation will use **malloc( )** system call, which is a wrapper to **_malloc_zone_malloc** function, to allocate heap memory. Eventually, it might seem weird when compared to other systems, but in macOS allocations via malloc function are performed inside **malloc zones**, such as an array of objects with the same size. In other words, the use of zones is a way to put allocations (objects) of the same size and type together at the same place.

From this point onward, it could be interesting to clone the libmalloc repository from the Apple Distribution macOS project:

- **git clone --recursive https://github.com/apple-oss-distributions/libmalloc.**

There is a standard **default zone** named **DefaultMallocZone** (from **libmalloc\src\malloc.c**) that is also known as **scalable zone** because it can scale to different allocation sizes created by the system. Additionally, **custom zones** may be created, and each of these zones have their own blocks and free memory blocks. This way, zones can be classified based on the requested size for memory to be allocated, and the system makes available the following categories, whose sizes can be checked in **libmalloc\src\magazine_zone.h** file:

- **Nano Zone:** this zone is dedicated to quite small allocations, and its allocations are smaller than 256 bytes, and quantum size (**block size**, which is the minimal allocated data) of 16 bytes. Nonetheless, it is not enabled by default. Its structure is **nanozone_s**, which is defined in **libmalloc\src\nanozone.h** file.

- **Tiny Zone**: this zone is dedicated to small allocations, and it has 1 MB with allocations (< 1008 bytes). The quantum size is 16 byes. An allocation chunk is the minimum and basic structure that is used by the malloc function, and such chunk is composed of multiple blocks, where each used chunk has metadata information, but unused chunks do not have such metadata information. Additionally, when the chunk is free (and belongs to a free list, and there are 64 free lists), it has a couple of pointers (previous and next) that points to the previous and next free chunks, respectively. As expected, and based on other examples of exploitation, such pointers are protected by a checksum and they are also shifted pointers, which makes it harder to modify them.

- **Small Zone:** this zone is dedicated to small-sized allocations, and it has 8 MB with allocations that start in 1009 bytes and goes up to 32 KB. The quantum size is 512 bytes. There are also the same previous and next pointers to freed chunks (they work as a doubly-linked list, but they are used as raw pointers, and a new component (a byte) is added to the checksum. Another difference is that a page-aligned free chunk is named as "oob free chunk". They do not have metadata associated and make part of an array. The region itself stores metadata at its end.

- **Medium Zone:** this zone is dedicated to medium-sized allocation, and its allocations start with 32 KB and have a maximum of 8192 KB. The quantum size is 32 KB.

- **Large Zone:** this zone is dedicated to large allocation, and it has over 8MB with allocations larger than 15 KB, or larger than 127KB if memory is +1GB.

Such allocations (mainly tiny and small ones, which will be very used) within each region must be managed, and that is the reason to use a structure named **magazine**, which works like a cache and has a series of fields for managing its structure. As we will see again later, a **rack** (like a partition) is composed of one or more magazines, and there is a link between racks and each magazine. There is also **zone depot**, which holds a list of full and empty magazines (explained in future articles). For now, you can imagine something as process **memory → rack → magazine → regions (zones) → blocks (quantum)**. It is the rack that holds the magazine type and another metadata that is the number of regions that make part of the rack.

During the calling of **malloc function** is size parameter that is the first selector to decide what region will be used. However, as already explained, these regions work like a cache. If the requested size is not found in this cache, the respective free lists will be verified. If the chosen chunk is a regular chunk then it is taken, but if the chosen chunk comes from the **oob chunk array**, so malloc takes the next chunk. This procedure is the same for tiny and small zones except by the checksum because it works with raw pointer in the small zone context.

All associated values of different zones can be checked in **libmalloc\src\thresholds.h** file. Furthermore, there are also multiple other zones defined for mechanisms and applications, as well as **purgeable zones**, where allocated pages can be freed when it is necessary (low memory), but it is not the moment to comment about it (check **libmalloc\src\purgeable_malloc.c** and related files). Anyway, zones in general can be listed by executing the following command (use **sudo** command or **root** user to run it), and **zone_info** structure (**osfmk\mach_debug\zone_info.h**).

The next two figures show part of the source code where zone thresholds are shown and also a list of different zones created in this macOS installation:

```
27: /*
28:  * The actual threshold boundaries between allocators. These boundaries are
29:  * where the next allocator will take over and they are *inclusive* of the
30:  * limit value.  That is, a TINY limit of X implies that an X-byte
31:  * allocation will come from TINY.
32:  *
33:  * The LARGE allocator cuts in at whatever the last boundary limit is. So, when
34:  * the medium allocator is compiled out, or not engaged, then large starts at
35:  * the limit of SMALL.
36:  */
37: #if MALLOC_TARGET_64BIT
38: #define TINY_LIMIT_THRESHOLD (1008)
39: #else // MALLOC_TARGET_64BIT
40: #define TINY_LIMIT_THRESHOLD (496)
41: #endif // MALLOC_TARGET_64BIT
42:
43: #if MALLOC_TARGET_IOS
44: #define SMALL_LIMIT_THRESHOLD (15 * 1024)
45: #else // MALLOC_TARGET_IOS
46: #define SMALL_LIMIT_THRESHOLD (32 * 1024)
47: #endif // MALLOC_TARGET_IOS
48: #define MEDIUM_LIMIT_THRESHOLD (8 * 1024 * 1024)
49:
50: /*
51:  * Tiny region size definitions; these are split into quanta of 16 bytes,
52:  * 64504 blocks is the magical value of how many quanta we can fit in a 1mb
53:  * region including the region trailer and metadata.
54:  *
55:  * XXX Although much of the tiny implementation handles (msize == 0) values as
56:  * 65536, this configuration of NUM_TINY_BLOCKS makes that value impossible to
57:  * reach.  That should be cleaned up, as it would greatly simplify msize
58:  * handling.
59:  */
60: #define SHIFT_TINY_QUANTUM 4ull
61: #define SHIFT_TINY_CEIL_BLOCKS 16 // ceil(log2(NUM_TINY_BLOCKS))
62: #define TINY_QUANTUM (1 << SHIFT_TINY_QUANTUM)
63: #define NUM_TINY_BLOCKS 64504
64: #define NUM_TINY_CEIL_BLOCKS (1 << SHIFT_TINY_CEIL_BLOCKS)
65: #define NUM_TINY_SLOTS (TINY_LIMIT_THRESHOLD >> SHIFT_TINY_QUANTUM)
66:
67: #if MALLOC_TARGET_64BIT
68: #define TINY_BITMAP_RANGE_LIMIT 63
69: #else
70: #define TINY_BITMAP_RANGE_LIMIT 31
71: #endif
72:
73: /*
74:  * Small region size definitions.
75:  *
76:  * We can only represent up to 1<<15 for msize; but we choose to stay
77:  * even below that to avoid the convention msize=0 => msize = (1<<15)
78:  */
79: #define SHIFT_SMALL_QUANTUM (SHIFT_TINY_QUANTUM + 5) // 9
80: #define SHIFT_SMALL_CEIL_BLOCKS 14 // ceil(log2(NUM_SMALL_BLOCKs))
80: #define SHIFT_SMALL_CEIL_BLOCKS 14 // ceil(log2(NUM_SMALL_BLOCKs))
81: #define SMALL_QUANTUM (1 << SHIFT_SMALL_QUANTUM) // 512 bytes
82: #define SMALL_BLOCKS_ALIGN (SHIFT_SMALL_CEIL_BLOCKS + SHIFT_SMALL_QUANTUM) // 23
83: #define NUM_SMALL_BLOCKS 16319
84: #define NUM_SMALL_CEIL_BLOCKS (1 << SHIFT_SMALL_CEIL_BLOCKS)
85: #define NUM_SMALL_SLOTS (SMALL_LIMIT_THRESHOLD >> SHIFT_SMALL_QUANTUM)
86:
87: /*
88:  * Medium region size definitions.
89:  *
90:  * We can only represent up to 1<<15 for msize; but we choose to stay
91:  * even below that to avoid the convention msize=0 => msize = (1<<15)
92:  */
93: #define SHIFT_MEDIUM_QUANTUM (SHIFT_SMALL_QUANTUM + 6) // 15
94: #define SHIFT_MEDIUM_CEIL_BLOCKS 12ull // ceil(log2(NUM_MEDIUM_BLOCKS))
95: #define MEDIUM_QUANTUM ((uint64_t)(1 << SHIFT_MEDIUM_QUANTUM)) // 32kbytes
96: #define MEDIUM_BLOCKS_ALIGN (SHIFT_MEDIUM_CEIL_BLOCKS + SHIFT_MEDIUM_QUANTUM) // 27
97: #define NUM_MEDIUM_BLOCKS 4095
98: #define NUM_MEDIUM_CEIL_BLOCKS (1ull << SHIFT_MEDIUM_CEIL_BLOCKS)
99: #define NUM_MEDIUM_SLOTS (MEDIUM_LIMIT_THRESHOLD >> SHIFT_MEDIUM_QUANTUM)
100: #define MEDIUM_ACTIVATION_THRESHOLD (32ull * 1024 * 1024 * 1024)
101: #define MEDIUM_CONDITIONAL_MADVISE_LIMIT (2 * 1024 * 1024)
102: #define MEDIUM_MADVISE_SHIFT 4
103: #define MEDIUM_MADVISE_MIN ((3 * 1024 * 1024) / 2) // 1.5 megabytes
```

[Figure 56]: libmalloc\src\attributes.h file (truncated)

```
Mac:~ root# zprint | head -25
                                  elem        cur        max        cur        max        cur  alloc  alloc
zone name                         size       size       size      #elts      #elts      inuse   size  count
--------------------------------------------------------------------------------------------------------------
vm.permanent                         1        68K        68K      69632      69632      64875     4K   4096
vm.permanent.percpu                  4        96K        96K      24576      24576      22840    16K   4096
threads_ro                          96       220K       220K       2346       2346       1624     4K     42  C
MAC.Labels                          64       660K       660K      10560      10560       9930     4K     64  C
proc_ro                            128        76K        88K        608        704        565     4K     32  C
cred                               144       220K       268K       1564       1905       1363     4K     28  C
cs_blob.zone                       184       176K       184K        979       1024        881     4K     22  C
sandbox.sandboxes                   88        48K        48K        558        558        465     4K     46  C
sandbox.profiles                   144        52K        52K        369        369        337     4K     28  C
sandbox.protobox.index              40         4K         4K        102        102          1     4K    102  C
sandbox.syscallmasks                72         8K        12K        113        170         93     4K     56  C
amfi.osentitlements                120       120K       120K       1024       1024        941     4K     34  C
pmap                               384       204K       216K        544        576        534    12K     32  C
maps                               168        96K       108K        585        658        557    12K     73  C
VM.map.entries                      80      4952K      5116K      63385      65484      57512     4K     51  C
VM.map.holes                        32       252K       256K       8064       8192       7331     4K    128  C
VM.map.copies                       72        88K        96K       1251       1365        728     8K    113  C
vm.pages.array                      56     55296K     55296K    1011126    1011126     978578     4K     73 XC
ipc.ports                          152     13312K     13312K      89680      89680      88461    16K    107  C
ipc.port.sets                       56        60K        64K       1097       1170       1087     4K     73  C
ipc.kmsgs                          256       688K       708K       2752       2832       1606     4K     16  C
ipc.vouchers                        56        28K        28K        512        512        258     4K     73  C
Mac:~ root#
```

[Figure 57]: Zone list (truncated)

There is also a reference to **nanov2_zone**, which is based on arena allocation, and it is defined in **libmalloc\src\nanov2_zone.h** file where we learn that it works with a 16 KB block size, the arena size is 64MB and the region size is 512 MB. Within the arena, there are 4096 blocks, and a region can have up to 8 arenas. The **nano2v zone** is represented by **nanozonev2_s structure**.

There are a series of malloc functions associated with zones: **malloc_create_zone, malloc_destroy_zone, malloc_zone_malloc, malloc_zone_calloc, malloc_zone_relloc, malloc_zone_free, malloc_zone_register, malloc_zone_unregister**, and other ones. The **zone structure** (**osfmk\kern\zalloc_internal.h**) itself has the following content:

```
227: struct zone {
228:     /*
229:      * Readonly / rarely written fields
230:      */
231:
232:     /*
233:      * The first 4 fields match a zone_view.
234:      *
235:      * z_self points back to the zone when the zone is initialized,
236:      * or is NULL else.
237:      */
238:     struct zone         *z_self;
239:     zone_stats_t         z_stats;
240:     const char          *z_name;
241:     struct zone_view    *z_views;
242:     struct zone_expand *z_expander;
243:
244:     uint64_t             z_quo_magic;
245:     uint32_t             z_align_magic;
246:     uint16_t             z_elem_size;
247:     uint16_t             z_elem_offs;
248:     uint16_t             z_chunk_pages;
249:     uint16_t             z_chunk_elems;
```

[Figure 58]: Zone structure (truncated)

To understand how a scalable zone (default zone) works and being able to correlate further details related to memory management that will be presented later, readers should adopt the following memory organization:

- The process's memory is divided into **three parts**.
- The **first part** is dedicated to **tiny allocations**.
- The **second part** is dedicated to **small allocations**.
- The **third part** is dedicated to **larger allocations**.
- These parts are known as **racks**. (check **libmalloc\src\magazine_rack.c** file)
- The **first two racks** (for tiny and small allocations) are populated with magazines (**magazine_s structure**, which is defined in **libmalloc\src\magazine_zone.h**), where **each magazine is composed of multiple regions** according to the respective allocations (1MB for tiny allocations and 8MB for small allocations).
- All **magazines** contain a **metadata region** (usually protected, mainly the pointers that are checked during coalescing operations) at the **end of the region**, where information like **previous pointer, forward pointer and size are saved**. Note that metadata is specific to the rack.
- Each region is subdivided into **quantum**, where **each tiny allocation contains multiple 16 bytes quantum,** and each **small allocation contains multiple 512 bytes quantum**.

The composition shown above represents how memory is organized in magazines and subdivisions. Furthermore, in the past, magazines were created to provide CPU caches (one magazine per core), and it makes sense because when an allocation is freed, the respective block is cached exactly in the magazine and, eventually, it replaces the old cached entries causing coalescing of successive freed blocks, and such a coalesced block is added into the free list, which is managed through previous and next pointers (both should be protected, by the way) in a doubly-linked list. As a result, next time that a block is allocated, the requested block size will be searched in the cache, free list, end of the region and, at the limit, a new region will be allocated to attend the request. At the end of the day, we are looking for contiguous and non-randomized ASLR for exploitation, and in determined contexts, there is a possibility here.

There are a series of environment variables (credits: all of them pulled from malloc man page) that can change the behavior of allocation-related functions and allow us to be tracking what is really happening:

- **MallocDebugReport:** it specifies where messages are written.
- **MallocGuardEdges:** it adds a guard page before and after each large block.
- **MallocDoNotProtectPrelude:** it does not add a guard page before large blocks.
- **MallocDoNotProtectPostlude:** it does not add a guard page after large blocks.
- **MallocStackLogging:** it records all allocation and deallocation events to an on-disk log, along with stacks, so that tools like leaks(1) and **malloc_history**(1) can be used. Possible values: default, vm (records only allocation of virtual memory regions), malloc (it records only allocations via malloc and respective interfaces) and lite (it records only the current allocation).
- **MallocStackLoggingDirectory:** it records stack logs to the directory specified instead of saving them to the default location (/tmp).
- **MallocScribble:** it fills memory that has been allocated with 0xaa bytes.
  memory will fail. It also fills memory that has been deallocated with 0x55 bytes.
- **MallocCheckHeapStart <s>:** it specifies the number of allocations <s> to wait before beginning periodic heap checks every <n> as specified by MallocCheckHeapEach.

- **MallocCheckHeapEach <n>:** it runs a consistency check on the heap every <n> operations. MallocCheckHeapEach is only meaningful if MallocCheckHeapStart is also set.
- **MallocCheckHeapSleep <t>:** it sets the number of seconds to sleep (waiting for a debugger to attach) when MallocCheckHeapStart is set, and a heap corruption is detected. The default is 100 seconds.
- **MallocCheckHeapAbort <b>:** when MallocCheckHeapStart is set and this is set to a non-zero value, causes abort(3) to be called if a heap corruption is detected, instead of any sleeping.
- **MallocErrorAbort:** it causes abort(3) to be called if an error was encountered in malloc(3) or free(3), such as a calling free(3) on a pointer previously freed.
- **MallocCorruptionAbort:** it is similar to MallocErrorAbort but will not abort in out of memory conditions, making it more useful to catch only those errors which will cause memory corruption. MallocCorruptionAbort is always set on 64-bit processes.
- **MallocZeroOnFree:** Starting in macOS 13, iOS 16.1 and aligned releases, free(3) fully zeroes many blocks immediately. This may expose some previously-silent bugs in existing applications. In particular, read-after-free bugs may now observe zeroes instead of the previous content of an allocation, and write-after-free bugs may cause calloc(3) to return non-zero memory. MallocZeroOnFree can be set to 0 or 1 to explicitly disable or enable this zeroing behavior to aid in diagnosing such bugs.
- **MallocCheckZeroOnFreeCorruption**: when zero-on-free behavior is active, this environment variable can be set to 1 to cause the allocator to check that the free block chosen for a given allocation remained fully zeroed and was not corrupted by any invalid use-after-free writes. If corruption is detected, the allocator will abort the operation.
- **MallocHelp:** it prints a list of environment variables that are paid heed to by the allocation-related functions, along with short descriptions.

I have brough up the magazine freelist array topic, and It is large enough to hold freed blocks from diverse kinds of allocation (tiny, small, and medium). Additionally, there is a special place within the list for coalesced regions that are larger than the maximum allocation size of the designed allocator. As readers notice, there is a series of mechanisms to guarantee the decent work and performance of the memory system, which is compressed (run **vm_stat | grep compr to** check it).

Under **security aspects**, we can make comments on additional points that might be useful:

- **Freed elements** are tracked by the **free list**.
- Very probably these freed elements from a zone (by garbage collector) will cause holes, which later will suffer **coalescing** and a reorganization of such elements that, at the end, will be **re-used in another zone.**
- Obviously, all released elements from a zone were **related to a previous object**, and it can represent a security risk because, similarly to other targets in Windows and Linux contexts, an attacker could try to artificially **decrease reference counting to zero of** a given allocated object to **change its status to free while it is still used**. This object would be **freed and later re-used by a new allocation (use-after-free), so it would be possible to write anything in memory of the new object.**
- Over time, protections have been added to introduce **allocation randomization through a kernel slide (printed as kernel slid) in the kernel base** to prevent any fixed address, also **included guard**

protection in both sides of a freed element and **these protections (a kind of a token or cookie) are re-checked when the element memory is re-used**, so avoiding any manipulation.
- Allocations are contiguous and not randomized under certain conditions.
- A region may have allocations of assorted sizes under certain restrictions.
- Another well-known technique is creating a chunk and, playing with the size meta-information , try to alter the context of another (the next) chunk.
- Even having a favorable heap layout, different protections implemented over time can make real exploitation hard.

Zalloc function (very hardened in current days) is used to allocate memory from a predefined memory zone, and there are its variants like **zalloc_noblock** that never blocks. The default zone is managed by a **zone allocator**, which is great for static zone sizes, but it might be better to dynamic allocation lengths. Other interesting points:

- In general, memory can be allocated by using functions such **as zalloc, zalloc_percpu, zalloc_permanent and kalloc**.
- The **zone allocator** keeps a **bitmap** that indicates what **elements are allocater or free**.
- **Zone tagging** allows us to **tag** and **account allocations**.
- **Probabilistic guard zalloc** is an alternative protection.
- **Zone zeroing** allows the possibility of having **all allocations from a zone zeroed when allocated blocks become free**.
- **Zone leak debugging code** to find leaks from a zone.
- **Zone corruption logging** to find the source of a zone corruption.
- **Zone for logging**.
- In zones, a **random grow of memory** prevents **a precise prediction of the heap** and, consequently, making it harder to deploy an **out-of-boundary exploitation technique** to compromise a zone page and cause side effect (arbitrary writing) over another one.
- **Guards** added after each chunk to mitigate out-of-boundary exploitation, mainly in zones allocated by kalloc function. Thus, this measure can **reduce the effectiveness of the FENG-SHUI technique (there are newer defenses against it)** because it is highly likely to find a guard.
- A series of **out-of-boundary exploitation techniques** are based on **precise interleaving of specific types** and, if this kind of interleaving is no longer possible then such techniques are broken.
- **Chunk lengths are randomized** and, additionally, guard pages are also used with new chunks.

All of these exposed points deserve to be detailed and extended because they are rich topics and offer an endless number of learning opportunities. In this article I am not going to exploit them (it is only the first article of this macOS series), but I will eventually return to the theme. To finish this section, a couple of brief comments about **two mechanisms that are used to manage memory pressure** follow:

- **Memorystatus** mechanism (from macOS) try to evaluate memory pressure, which is caused by a series of events and measures such as amount of wired memory (resident memory), file cache usage, free memory, and other items. When there is a shortage of memory then the mechanism starts its action and gradually terminates application.

- **Jetsam mechanism** (from iOS) shows a similar behavior to **memorystatus**, but it is much more aggressive to free memory because mobile devices have less resources.

7.4. IPC mechanisms

IPC (Inter-Process Communication) is a mechanism that enables processes to communicate with each other by **exchanging messages** among running processes using **synchronous** (semaphores and mutexes, for example) or **asynchronous** techniques. The most common goal is to share information about the processes involved and, at the same time, obtain a gain of performance. The portfolio of IPC mechanisms is vast and include **socket, Apple events, XPC, shared memory** and **shared file**, and all of them offer solutions to specific contexts and situations.

Shared memory is a simple form of IPC, and terribly similar to any other operating system, and basically an exchange of contents occurs between a source and a target processes. Example of shared memory functions are available such as **shmget** (create a segment of memory), **shmat** (attach the shared memory segment to the address of the source process) and their helpers such as **shmdt** (detaches the memory segment) and **shmctl** (it is responsible for controlling operations based on shared memory).

Mach port, as we have learned earlier this article, plays an important role on IPC because it provides all necessary foundation to IPC techniques such as **semaphores** (usually deployed to control and synchronize the usage of resources), **message queues**, **notification**, **locks** (similar to mutexes) and **RPC** (Remote Procedure Call), for example.

To understand such communication methods, we should remember about Mach concepts such as:

- **tasks** are similar to processes in BSD systems, and they contain one or more threads, being the goal of a task to provide resources to its threads.
- **Mach port** is the main communication method between tasks.
- tasks hold **port rights** to access ports because **a port can be accessed only through a given right**.
- there are distinct types of rights, and **the objective of having a port right is to access or even modify an object, which is a resource accessed by a port**.
- tasks can **copy or even move port rights** between them using IPC.
- there are different port rights such as **send right** (a task is allowed to send messages to a given port), **receive right** (a task can receive messages sent to the port), **send-once** (a task is allowed to send one and only one message to a the given port) and **port set right** (a port set is an array of rights that allows a task to list on multiple ports).
- the **port communication** is one-way direction.
- the message exchange happens through **message queues** in an asynchronous way.
- the entire communication of Mach resources (except in memory case) it through **Mach ports**.

In terms of Mach ports, readers will find functions such as **mach_msg, mach_port_allocate, mach_port_insert_right** and **bootstrap_register** in terms of Mach layer, but you will be managing high level functions from the **Foundation** and **Core Foundation** frameworks too.

Mach is a natural candidate for anything related to communication because its working is based on message exchanges and fundamentally for the kernel. There are a series of data types related to IPC such as **ipc_object, ipc_entry_t, ipc_set_t**, and also structures such as **ipc_port, ipc_mqueue** and **ipc_space** (and other ones), and such types are the foundation of the mechanism as a whole.

Such IPC data type can be examined in **osfmk\ipc\ipc_types.h** file, as shown below:

```
/*
 * Define Basic IPC types available to callers.
 * These are not intended to be used directly, but
 * are used to define other types available through
 * port.h and mach_types.h for in-kernel entities.
 */

#ifndef _IPC_IPC_TYPES_H_
#define _IPC_IPC_TYPES_H_

#include <mach/port.h>
#include <mach/message.h>
#include <mach/mach_types.h>

#ifdef  MACH_KERNEL_PRIVATE

typedef natural_t ipc_table_index_t;/* index into tables */
typedef natural_t ipc_table_elems_t;/* size of tables */
typedef natural_t ipc_entry_bits_t;
typedef ipc_table_elems_t ipc_entry_num_t;  /* number of entries */
typedef ipc_table_index_t ipc_port_request_index_t;

typedef mach_port_name_t mach_port_index_t;         /* index values */
typedef mach_port_name_t mach_port_gen_t;           /* generation numbers */

typedef struct ipc_entry *ipc_entry_t;

typedef struct ipc_table_size *ipc_table_size_t;
typedef struct ipc_port_request *ipc_port_request_t;
typedef struct ipc_pset *ipc_pset_t;
typedef struct ipc_kmsg *ipc_kmsg_t;
typedef uint8_t sync_qos_count_t;

typedef uint64_t ipc_label_t;
#define IPC_LABEL_NONE          ((ipc_label_t)0x0000)
#define IPC_LABEL_DEXT          ((ipc_label_t)0x0001)
#define IPC_LABEL_PLATFORM      ((ipc_label_t)0x0002)
#define IPC_LABEL_SPECIAL       ((ipc_label_t)0x0003)
#define IPC_LABEL_SPACE_MASK    ((ipc_label_t)0x00ff)
```

[Figure 59]: IPC types (truncated)

The file **osfmk\ipc\ipc_port.h** contains a series of structures, and one of them is **ipc_port**, as shown below:

```
struct ipc_port {
    struct ipc_object               ip_object;
    union {
        /*
         * The waitq_eventmask field is only used on the global queues.
         * We hence repurpose all those bits for our own use.
         *
         * Note: if too many bits are added, compilation will fail
         *       with errors about "negative bitfield sizes"
         */
        WAITQ_FLAGS(ip_waitq
            , ip_fullwaiters:1            /* Whether there are senders blocked on a full queue */
            , ip_sprequests:1             /* send-possible requests outstanding */
            , ip_spimportant:1            /* ... at least one is importance donating */
            , ip_impdonation:1            /* port supports importance donation */
            , ip_tempowner:1              /* dont give donations to current receiver */
            , ip_guarded:1                /* port guarded (use context value as guard) */
            , ip_strict_guard:1           /* Strict guarding; Prevents user manipulation of context values directly */
            , ip_specialreply:1           /* port is a special reply port */
            , ip_sync_link_state:3        /* link the port to destination port/ Workloop */
            , ip_sync_bootstrap_checkin:1 /* port part of sync bootstrap checkin, push on thread doing the checkin */
            , ip_immovable_receive:1      /* the receive right cannot be moved out of a space, until it is destroyed */
            , ip_immovable_send:1         /* No send(once) rights to this port can be moved out of a space, never unset *
            , ip_no_grant:1               /* Port wont accept complex messages containing (ool) port descriptors */
            , ip_tg_block_tracking:1      /* Track blocking relationship between thread groups during sync IPC */
            , ip_pinned:1                 /* Can't deallocate the last send right from a space while the bit is set */
            , ip_service_port:1           /* port is a service port */
            , ip_has_watchport:1          /* port has an exec watchport */
            , ip_kernel_iotier_override:2 /* kernel iotier override */
            , ip_kernel_qos_override:3    /* kernel qos override */
            , ip_reply_port_semantics:3   /* reply port defense in depth type */
            );
        struct waitq            ip_waitq;
    } « end {anon_union} » ;
```

[Figure 60]: osfmk\ipc\ipc_port.h (truncated)

Although I have quickly explained about Mach ports, another important concept is a **message descriptor**, which provides a viable method to include one or more port rights in a message as a sender. Additionally, **port set descriptors** hold a number of ports that will be included in the message and also allow to provide a virtual address space region. At the same time, it is excellent that the possibility of copying rights and virtual address space regions to the receiving port, if the target does not check what is being copied then vulnerable might arise. Following the same line, port set descriptors are also used to cause heap memory manipulation. There are multiple source-files that readers can learn a lot on IPC details because they are really well-commented, and some of them are (both .h and .c extensions**) osfmk\ipc\ipc_object, osfmk\mach_debug\ipc_info, osfmk\ipc\ipc_space, osfmk\ipc\ipc_notify and osfmk\ipc\ipc_mqueue** (once again, add extensions .h and .c for mentioned files).

Likely sockets, in terms of high-level communication, are the most known when discussing on inter-process communication because such communication happens using packets and there is a well-established portfolio of network APIs as **POSIX, CFNetwork framework, NSStream and CFStream** and **C Networking APIs** from Core Foundation. Anyway, sockets are great for one-to-one communication, but most of the time are used for a low amount. On the other hand, Socket is a universal technology among platforms.

Apple offers its own IPC mechanism for communication that is **Apple Events**, which as suitable for GUI applications on macOS in communications inside the system or even across different systems and usually deployed for requesting or providing other applications with services. In a general way, applications register **callback routines** in the **Apple Event Manager** and soon the application receives an Apple event it will dispatch such event to the more appropriate handlers via the same **Apple Event Manager**. To use and interact with this manager, one of the alternatives is to use **AppleScript**, which is used by Cocoa applications, for example. Additionally, **AppleScript** is a valuable, simple, and accessible resource to macOS users and power users. Unfortunately, not all applications are scriptable as we already could imagine.

I have quickly mentioned notifications that are a key component because they contain information about events and when an event happens then it is published to the **notification center,** which is responsible for sending (via **broadcasts**) such notifications to all **registered objects**, as mentioned previously. Similar to other operating systems, notifications do not need to be released immediately, and there could be a delay in this transmission (in this case, **asynchronously**). The object, which eventually can have a name that serves as an identification, registers itself to the notification center and can exchange information with other objects through messages that are built in **NSNotification objects** . Additionally, a **notification object** not only receives notifications (messages), but it is also able to publish (post) a notification, and usually such registration comes from an application, even it is able to receive notification messages from other notification centers once it has done a registration to a **distributed notification center**.

Moving to a different IPC mechanism, certainly one of most discussed inter-process communication topics under the security context (sometimes related to existing vulnerabilities) is **XPC** because it is a powerful, stable and enhanced mechanism that makes easier the communication through messages between application's components and also makes it an appropriate framework for offering services, which are implemented as **Bundles** -- **xpc_bundle_t**) and, similarly to other Apple mechanisms, there are two levels of APIs such low-level and high-level (it uses **NSXPCConnection**: https://developer.apple.com/documentation/foundation/nsxpcconnection) and **NSXPCListener**: https://developer.apple.com/documentation/foundation/nsxpclistener), and in the latter case programming languages such as Objective-C and Swift can be used to generate applications.

As a good IPC mechanism, XPC offers effective inter-process communication through a listener (server) and client peers (including remote hosts). XPC framework has security protections to avoid exploitation and, for example, each XPC service is **managed by launchd and gcd** and is also **sandboxed**, which provides the due isolation that, with the help offered by entitlements, mitigate problems related to sandbox escape and elevation of privilege. Unfortunately, eventual software vulnerabilities have come up over the years, and even sandboxes escaping techniques have been revealed (probably there are many unrevealed exploits in the wild). Additionally, the core of XPC (libxpc) is not open-sourced (readers can verify Apple open-source projects on https://opensource.apple.com/releases/), but there are good projects on GitHub from professional that started doing a reimplementation of it. At the same way, there have been a few vulnerabilities related to it such as **CVE-2020-9971** and **CVE-2021-30652**, for example.

The high-level communication is quite accessible because it depends on a connection (from both sides), an interface that works as a contract and that exposes all available methods, an XPC service that accepts eventual connections and the message itself that is used to exchange information in a bidirectional and asynchronous way. The message data involved in XPC communication is formatted and encoded as a dictionary (**xpc_dictionary_t**), which establishes a kind of protection against attacks like type confusion.

There is a low-level interface known as XPC Service APIs which is very interesting, with many functions starting with **__xpc_** prefix, and that handles a wide range of specific XPC types (even though XPC does not have a defined schema) and also prefixed at the same form (**__xpc_type_int64**, **__xpc_type_string**, **__xpc_type_array, __xpc_type__data** and many others), as shown below:

```
alexandreborges@Mac ~ % ARCH=x86_64 jtool2 -S libxpc.dylib | grep D | grep xpc_type
00000000000447d0 D __xpc_type_activity
0000000000044460 D __xpc_type_array
0000000000044000 D __xpc_type_base
00000000000440a0 D __xpc_type_bool
00000000000446e0 D __xpc_type_bundle
0000000000044550 D __xpc_type_connection
0000000000044280 D __xpc_type_data
0000000000044230 D __xpc_type_date
00000000000444b0 D __xpc_type_dictionary
0000000000044190 D __xpc_type_double
00000000000445a0 D __xpc_type_endpoint
0000000000044500 D __xpc_type_error
0000000000044370 D __xpc_type_fd
0000000000044820 D __xpc_type_file_transfer
00000000000440f0 D __xpc_type_int64
0000000000044690 D __xpc_type_mach_recv
0000000000044410 D __xpc_type_mach_send
0000000000044870 D __xpc_type_mach_send_once
0000000000044050 D __xpc_type_null
0000000000044640 D __xpc_type_pipe
00000000000441e0 D __xpc_type_pointer
00000000000448c0 D __xpc_type_rich_error
00000000000445f0 D __xpc_type_serializer
0000000000044730 D __xpc_type_service
0000000000044780 D __xpc_type_service_instance
0000000000044a00 D __xpc_type_session
00000000000443c0 D __xpc_type_shmem
00000000000442d0 D __xpc_type_string
0000000000044140 D __xpc_type_uint64
0000000000044320 D __xpc_type_uuid
alexandreborges@Mac ~ %
```

[Figure 61]: listing XPC data types

The option **-S** is used to list symbols (similar to **nm** command) and the "**-D**" in the grep is because we are interested in **global symbols from data segment**. For sure, you can use the same approach to get global symbols from the text section, which most of the time represent routines:

```
alexandreborges@Mac ~ % ARCH=x86_64 jtool2 -S libxpc.dylib | grep T | grep __xpc_
000000000000cb34 T ___xpc_connection_set_logging
0000000000007fda T __xpc_bool_create_distinct
0000000000008004 T __xpc_bool_set_value
000000000000af83 T __xpc_connection_create_internal_listener
000000000000b577 T __xpc_connection_get_parent_4test
000000000000b573 T __xpc_connection_get_recvp_4test
000000000000bff2 T __xpc_connection_set_event_handler_f
000000000000ed5b T __xpc_data_set_value
0000000000010f44 T __xpc_dictionary_create_reply_with_port
0000000000011625 T __xpc_dictionary_extract_mach_send
00000000000110fa T __xpc_dictionary_extract_reply_msg_id
0000000000011004 T __xpc_dictionary_extract_reply_port
00000000000110a4 T __xpc_dictionary_get_reply_msg_id
0000000000010f8f T __xpc_dictionary_set_remote_connection
000000000001105a T __xpc_dictionary_set_reply_msg_id
00000000000031d0 T __xpc_domain_routine
000000000001436e T __xpc_double_set_value
0000000000016070 T __xpc_fd_get_port
```

[Figure 62]: listing XPC symbols from text section

In terms of messages transferred using XPC, they formatted as dictionaries (as already stated) and are serialized. On the other side, in high-level, C APIs handle with objects, which are all derived from **xpc_object_t** type, and there are methods to manipulate distinguished types too.

To start a given XPC service, once the client makes a request for such XPC service, the **launchd** starts the **xpcproxyd**, which establishes due restrictions and launches the target XPC service. The process is accelerated by using a cache containing information on available XPC services. Additionally, there are **plist files** that hold the configuration about how each service is started, and such **plist files** are spread in different directories such as **/Library/LaunchDaemons** and **/System/Library/LaunchDaemons** folders, but there are other ones associated with agents (same directories, but the folder is **LaunchAgents**). If readers do a quick search on the system (the easiest command is **find / -name *.xpc > xpc_list.txt**) a lot of directories with extension .xpc, which makes part of an application, will come up and a quick filter as shown below shows many examples:

```
alexandreborges@Mac ~ % more xpc_list.txt | grep '/Library' | grep -v 'Simulator' | head -10
/Library/Apple/System/Library/CoreServices/XProtect.app/Contents/XPCServices/XProtectPluginService.xpc
/Library/Apple/System/Library/PrivateFrameworks/RemotePairing.framework/Versions/A/XPCServices/remotepairingd.
xpc
/Library/Apple/System/Library/PrivateFrameworks/MobileDevice.framework/Versions/A/XPCServices/MDRemoteServiceS
upport.xpc
/Library/Apple/System/Library/PrivateFrameworks/MobileDevice.framework/Versions/A/XPCServices/MDPrivilegedUSBS
upport.xpc
/Library/Developer/CommandLineTools/usr/lib/sourcekitd.framework/Versions/A/XPCServices/SourceKitService.xpc
/Library/Developer/CommandLineTools/SDKs/MacOSX12.3.sdk/System/Library/PrivateFrameworks/iWorkXPC.framework/Ve
rsions/A/XPCServices/iWorkFileFormat.xpc
/Library/Developer/CommandLineTools/Library/PrivateFrameworks/LLDB.framework/Versions/A/XPCServices/RootDebugg
ingXPCService.xpc
/Library/Developer/PrivateFrameworks/CoreDevice.framework/Versions/A/XPCServices/CoreDeviceService.xpc
/Library/Frameworks/CoreRepairCore.framework/Versions/A/XPCServices/CoreRepairCoreXPCService.xpc
/System/iOSSupport/System/Library/PrivateFrameworks/IMTranscoding.framework/XPCServices/IMTranscoderAgent.xpc
alexandreborges@Mac ~ %
```

[Figure 63]: Listing a few XPC related directories and files.

About the mentioned plist, you can verify one of the plist and see the key **MachServices** and the respective **service name** below:

```
alexandreborges@Mac LaunchDaemons % pwd
/System/Library/LaunchDaemons
alexandreborges@Mac LaunchDaemons % cat com.apple.mobile.keybagd.plist | head -25
<?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>EnablePressuredExit</key>
        <true/>
        <key>EnvironmentVariables</key>
        <dict>
                <key>__CFPREFERENCES_AVOID_DAEMON</key>
                <string>1</string>
                <key>__CF_USER_TEXT_ENCODING</key>
                <string>0x0:0:0</string>
        </dict>
        <key>Label</key>
        <string>com.apple.mobile.keybagd</string>
        <key>MachServices</key>
        <dict>
                <key>com.apple.mobile.keybagd.mach</key>
                <true/>
                <key>com.apple.mobile.keybagd.xpc</key>
                <true/>
        </dict>
        <key>RunAtLoad</key>
        <true/>
        <key>OnDemand</key>
alexandreborges@Mac LaunchDaemons %
```

[Figure 64]: Plist related to the service

The XPC service, which is managed by the **launchd**, is limited to a specific process because the **launchd process domain** that works as a namespace and stores the necessary information on the XPC service. This information must (or at least should) be retrieved and modified by the respective owner process. This behavior is shown below:

```
alexandreborges@Mac LaunchDaemons % ps aux | grep keybag
alexandreborges  27080   0.0  0.0 34121296    636 s001  R+    9:26PM   0:00.00 grep keybag
root               109   0.0  0.0 33658400   1876   ??  Ss   29Oct24   0:00.22 /usr/libexec/keybagd
 -t 15
alexandreborges@Mac LaunchDaemons %
alexandreborges@Mac LaunchDaemons % launchctl print pid/109
Could not print domain: 1: Operation not permitted
alexandreborges@Mac LaunchDaemons %
alexandreborges@Mac LaunchDaemons % whoami
alexandreborges
alexandreborges@Mac LaunchDaemons % sudo launchctl print pid/109
pid/109 = {
        type = pid
        handle = 109
        active count = 1
        on-demand count = 1
        originator = /usr/libexec
        creator = keybagd[109]
        creator euid = 0
        uniqueid = 109
        security context = {
                uid unset
                asid = 100000
        }
```

```
        xpc service paths = {
        }
        death port = 0xea03

        environment = {
                PATH => /usr/bin:/bin:/usr/sbin:/sbin
                __CF_USER_TEXT_ENCODING => 0x0:0:0
        }

        services = {
        }

        unmanaged processes = {
        }

        endpoints = {
        }

        task-special ports = {
                         0xb03 4        bootstrap  com.apple.xpc.launchd.domain.pid.keybagd.109
                        0x2403 9           access  (unknown)
        }


        properties =
}
```

[Figure 65]: Retrieving process domain information of a process.

If readers want to investigate the code, you can check the reversed code shown below and understand how the launchctl allows or not get or even modify the domain information:

```
23    v25 = 0LL;
24    v12 = sub_10000A423(*(char **)(a12 + 8), v15, &v25, a1, a2, a3, a4, v16, v17, a7, a8);
25    if ( !v12 )
26    {
27      v18 = sub_10000AADB(v15, 0x100000uLL);
28      v26[0] = 0LL;
29      if ( v25 )
30        v21 = sub_10000B89D(708, v15, v26, a1, a2, a3, a4, v19, v20, a7, a8);
31      else
32        v21 = sub_10000B8BA(828, v15, v26, a1, a2, a3, a4, v19, v20, a7, a8);
33      v12 = v21;
34      if ( v21 - 0x70 < 2 || v21 == 22 )
35      {
36        fwrite("Bad request.\n", 0xDuLL, 1uLL, __stderrp);
37      }
38      else if ( v21 )
39      {
40        v22 = __stderrp;
41        v23 = (const char *)xpc_strerror(v21);
42        fprintf(v22, "Could not print domain: %d: %s\n", v12, v23);
43      }
44      else
45      {
46        sub_10000AB3C(v26[0], v18, 0x100000uLL, __stdoutp);
47      }
48      sub_1000020F8((vm_address_t)v18, 0x100000uLL);
49      objc_release(v26[0]);
50    }
51    objc_release(v15);
52  }
53  return v12;
54}
```

[Figure 66]: Brief reversing of launchctl

Of course, it is not the moment to dig into reversing sessions, but a few observations **about launchd process domain** can be appropriate:

- **launchd** is a critical system process that is responsible for launching processes and managing system services and IPC communication (XPC), for example. Additionally, and as readers have already realized, it is managed by the **launchctl** application.
- **launchd** is able to talk to all processes running on system, and it also offers mechanisms to search and connect to such system services.
- the main structures associated with launchd are **domain, services,** and **endpoints**.
- each service has a collection of endpoints, and when a message is sent to one of these endpoints, the service is launched.
- the domain is responsible for managing the execution policy for a set of services.
- there are distinct kinds of domains:
    - **system**: it manages the root Mach bootstrap. As expected, root privileges are necessary to make any change.
    - **user:** it exists regardless of a logged-in user, but it does not exist on iOS.
    - **user-login:** it is created when a user logs-in at the GUI and, similarly to user domain, it does not exist on iOS.
    - **gui/session domain:** it is similar to login domain, but its identifier is based on the user. Thus, there are my resources shared between login and user domains.
    - **pid domain**: each process has a PID domain that consists of XPC services visible to the target process (it is exactly our case).

In a high-level view, services run on the top and use different abstraction layers such as MIG and XPC (for example) to communicate with the Mach layer (**XPC data type deserialization occurs between abstraction layer and Mach layer**), which exchanges Mach messages with the kernel. Security issues related to type confusion vulnerability should not happen, but may happen due to the existence of secure versions **xpc_dictionary_get_<type>** and **xpc_array_get_<type>**, and unsecure **xpc_dictionary_get_value** and **xpc_array_get_value** functions:

```
alexandreborges@Mac ~ % ARCH=x86_64 jtool2 -S libxpc.dylib | grep "xpc_dictionary_get"
000000000001132e t __xpc_dictionary_get_transaction
0000000000012c25 t __xpc_dictionary_get_importance_voucher
00000000000110a4 T __xpc_dictionary_get_reply_msg_id
0000000000012897 T _xpc_dictionary_get_array
0000000000011743 T _xpc_dictionary_get_audit_token
00000000000122e4 T _xpc_dictionary_get_bool
0000000000011398 T _xpc_dictionary_get_connection
0000000000011f65 T _xpc_dictionary_get_count
000000000001265f T _xpc_dictionary_get_data
00000000000125f7 T _xpc_dictionary_get_date
0000000000012867 T _xpc_dictionary_get_dictionary
000000000001257e T _xpc_dictionary_get_double
00000000000124ae T _xpc_dictionary_get_int64
00000000000114c3 T _xpc_dictionary_get_pointer
00000000000113c6 T _xpc_dictionary_get_remote_connection
000000000001270e T _xpc_dictionary_get_string
0000000000012516 T _xpc_dictionary_get_uint64
0000000000012782 T _xpc_dictionary_get_uuid
0000000000011526 T _xpc_dictionary_get_value
```

[Figure 67]: Different XPC service functions

I have mentioned about secure and unsecure versions of services functions because while secure versions check and ensure that the data entry is expected type, the unsecure ones do not perform such check.

In the next figures, which show the first two represent the secure version and the other one is an unsecure version, you can notice the following details:

- The secure version (**xpc_dictionary_get_int64(xpc_object_t xdict, const char *key)**) accepts a **xpc_object_t** type (which is a generic type), and soon at the beginning of the function it calls **_xpc_dictionary_look_up_fast** function.
- We can notice that the returned value (result) is checked to decide the next step, as can be visualized between the lines 20 and 32. If everything goes fine, so **xpc_int64_get_value** got called and the tagged pointer value (in case of pointers) will be managed.
- Within the **_xpc_dictionary_look_up_fast** function, we can see that the passed type is checked on line 17 (and 18) through **xpc_get_type** function and if it is not the expected type then the routine invokes **_xpc_api_misuse** with "Given object not of required type." message.
- If the type is correct then **_xpc_dictionary_look_up** is called and the function continues its processing normally even though the type is not the only checking that will be done.
- In the unsecure version, such type-checking does not exist, unfortunately, which is auspicious for **type confusion** vulnerabilities.
- To a real-world case and PoC, check https://project-zero.issues.chromium.org/issues/42452447. Additionally. it is always recommended to follow eventual messages.

```
1  int64_t __cdecl xpc_dictionary_get_int64(xpc_object_t xdict, const char *key)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5    v11[0] = 0LL;
6    memset(v10, 170, sizeof(v10));
7    result = _xpc_dictionary_look_up_fast(
8               xdict,
9               key,
10              12288,
11              v10,
12              (__m128)xmmword_2DD40,
13              v2,
14              v3,
15              v4,
16              v5,
17              v6,
18              v7,
19              v8);
20   if ( result )
21   {
22     if ( result == 0xDEADBEEFLL )
23     {
24       _xpc_int64_get_wire_value((__int64)v10, v11);
25       return v11[0];
26     }
27     else
28     {
29       return xpc_int64_get_value((xpc_object_t)result);
30     }
31   }
32   return result;
33 }
```

[Figure 68]: Secure XPC service function (part 1)

```
__int64 __fastcall _xpc_dictionary_look_up_fast(
        xpc_object_t object,
        const char *a2,
        int a3,
        char *a4,
        __m128 a5,
        __m128 a6,
        __m128 a7,
        __m128 a8,
        double a9,
        double a10,
        __m128 a11,
        __m128 a12)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  if ( xpc_get_type(object) != (xpc_type_t)&OBJC_CLASS___OS_xpc_error
    && xpc_get_type(object) != (xpc_type_t)&OBJC_CLASS___OS_xpc_dictionary )
  {
    _xpc_api_misuse("Given object not of required type.");
  }
  v14 = _xpc_dictionary_look_up((__int64)object, a2, a4);
  if ( !v14 )
    return 0LL;
  v21 = v14;
  if ( v14 == 0xDEADBEEFLL )
  {
    v22 = (int *)_xpc_graph_deserializer_read((__int64)a4, 4LL);
    if ( v22 )
      v23 = *v22;
```

[Figure 69]: Secure XPC service function (part 2)

Pseudocode-A

```
xpc_object_t __cdecl xpc_dictionary_get_array(xpc_object_t xdict, const char *key)
{
  xpc_object_t value; // rax
  void *v3; // rbx

  value = xpc_dictionary_get_value(xdict, key);
  if ( !value )
    return 0LL;
  v3 = value;
  if ( xpc_get_type(value) != (xpc_type_t)&OBJC_CLASS___OS_xpc_array )
    return 0LL;
  return v3;
}
```

```
0001689D _xpc_dictionary_get_array:5 (1289D)
```

Pseudocode-B

```
xpc_object_t __cdecl xpc_dictionary_get_value(xpc_object_t xdict, const char *key)
{
  if ( xpc_get_type(xdict) == (xpc_type_t)&OBJC_CLASS___OS_xpc_error
    || xpc_get_type(xdict) == (xpc_type_t)&OBJC_CLASS___OS_xpc_dictionary )
  {
    return (xpc_object_t)_xpc_dictionary_look_up((__int64)xdict, key, 0LL);
  }
  else
  {
    return 0LL;
  }
}
```

[Figure 70]: Unsecure XPC service functions

There are other IPC mechanisms::

- **CFMessagePort:** it provides a communication channel able to transmit data between processes or threads on the local machine. Methods such **CFMessagePortCreateLocal** (to create local messages), **CFMessageRemoteLocal** (to create remote messages), **CFMessagePortCreateRunLoopSource** (create a run loop source to listen messages) and **CFRunLoopAddSource** (add messages into a run loop) are normally used while programming using this mechanism.
- **Pasteboard:** it is a server shared by running applications, whose copied user data and other kinds of data can be transferred from one application to another. This mechanism is based on the **NSPasteboard objects**, which offers an interface to the server and all pasteboard operations.
- **Distributed Objects:** it uses RPC and exposes objects via proxy objects.
- **NSXPCConnection:** this method has already been mentioned previously, and its behavior is similar to distributed objects.

Personally, and that is a matter of opinion, I consider IPC one of the most important topics while researching security vulnerabilities macOS and iOS. Therefore, I would like to leave some additional words and include a concise review of important concepts that we have already seen.

- Mach IPC represent and offers the possibility of inter-communication **within the same system** and also via network.
- The **Mach port**, which represents tasks, threads, memory objects, hosts and other ones, should be treated as a kind of communication port (also interpreted as access point under the IPC context), which supports sending and receiving messages operations, and such operations allow tasks to place and retrieve messages from the internal queue.
- The access to a Mach port by tasks is controlled a **port right** (as well as known as port capability), which are predictably are send and receive rights (by the way, a port right is considered as a port name, in general). Attention here: even though a task has one or more threads, rights are granted task and not to its threads. As we have learned, rights can be:
    - **MACH_PORT_RIGHT_RECEIVE:** it can receive a message, and there could be multiple senders, but only one receiver.
    - **MACH_PORT_RIGHT_SEND:** it can send a message, and it has its references controlled.
    - **MACH_PORT_RIGHT_SEND_ONCE**: it can send a message once.
    - **MACH_PORT_RIGHT_PORT_SET:** it is given as a receive-right on multiple ports.
    - **MACH_PORT_RIGHT_DEAD_NAME:** this right can be a send or even a send-once right is not more valid because the associated port was destroyed.
- A **port set** is a set of receive rights.
- A nice concept on macOS/iOS is that a **Mach port right** (that is referred as **port name**, but they are different structures) can be transferred to another holder (receiver) through a message.
- A recommended file to check is **osfmk\ipc\mach_port.c** from the XNU source files because it contains a series of routines related to Mach port and also good comments. For example, ports can be allocated using **mach_port_allocate** function, destroyed using **mach_port_destroy** function and many other available routines.
- As we have introduced the Mach message concept, we have the **mach_msg** routine (a trap, actually) that is used to send or receive a message from a port (check **osfmk\man\mach_msg.html** and **libsyscall\mach\mach_msg.c** for its **description and definition) -- there are variants using this function)** as well **mach_msg_overwrite** function, which also send and receive a message.

```
516: kern_return_t
517: mach_port_allocate(
518:     ipc_space_t             space,
519:     mach_port_right_t       right,
520:     mach_port_name_t        *namep)
521: {
522:     kern_return_t           kr;
523:     mach_port_qos_t         qos = { };
524:
525:     kr = mach_port_allocate_full(space, right, MACH_PORT_NULL,
526:         &qos, namep);
527:     return kr;
528: }
```

```
151: /*
152:  *  Routine:    mach_msg
153:  *  Purpose:
154:  *      Send and/or receive a message.  If the message operation
155:  *      is interrupted, and the user did not request an indication
156:  *      of that fact, then restart the appropriate parts of the
157:  *              operation.
158:  */
159: mach_msg_return_t
160: mach_msg(
161:     mach_msg_header_t *msg,
162:     mach_msg_option_t option,
163:     mach_msg_size_t send_size,
164:     mach_msg_size_t rcv_size,
165:     mach_port_t rcv_name,
166:     mach_msg_timeout_t timeout,
167:     mach_port_t notify)
168: {
169:     return mach_msg_overwrite(msg, option, send_size, rcv_size, rcv_name,
170:                 timeout, notify, NULL, 0);
171: }
172:
173: /*
174:  *  Routine:    mach_msg_overwrite
175:  *  Purpose:
176:  *      Send and/or receive a message.  If the message operation
177:  *      is interrupted, and the user did not request an indication
178:  *      of that fact, then restart the appropriate parts of the
179:  *              operation.
180:  *
181:  *      Distinct send and receive buffers may be specified.  If
182:  *      no separate receive buffer is specified, the msg parameter
183:  *      will be used for both send and receive operations.
184:  *
185:  *      In addition to a distinct receive buffer, that buffer may
186:  *      already contain scatter control information to direct the
187:  *      receiving of the message.
188:  */
189: mach_msg_return_t
190: mach_msg_overwrite(
191:     mach_msg_header_t *msg,
192:     mach_msg_option_t option,
193:     mach_msg_size_t send_size,
194:     mach_msg_size_t rcv_limit,
195:     mach_port_t rcv_name,
196:     mach_msg_timeout_t timeout,
197:     mach_port_t notify,
198:     mach_msg_header_t *rcv_msg,
199:     __unused mach_msg_size_t rcv_scatter_size)
200: {
```

[Figure 71]: mach_port_allocate routine from osfmk\ipc\mach_port.c file, mach_msg and mach_msg_overwrite routines from libsyscall\mach\mach_msg.c file, respectively.

- In terms of Mach message, there are two types of messages that are **simple** and **complex**. In general, they are composed of a header (**mach_msg_header_t**), a body (**mach_msg_body_t** if the message type is complex) and a trailer (there are distinct types too, but the fundamental one is **mach_msg_trailer_t**). The **osfmk\mach\message.h** contains definition, but certainly the scheme below provides us with a better visualization:

```
typedef struct
{
        mach_msg_bits_t                 msgh_bits;
        mach_msg_size_t                 msgh_size;
        mach_port_t                   msgh_remote_port;
        mach_port_t                    msgh_local_port;
        mach_msg_size_t                msgh_reserved;
        mach_msg_id_t                     msgh_id;
} mach_msg_header_t;

typedef struct
{
        mach_msg_size_t         msgh_descriptor_count;
} mach_msg_body_t;

typedef struct
{
        mach_msg_trailer_type_t     msgh_trailer_type;
        mach_msg_trailer_size_t     msgh_trailer_size;
} mach_msg_trailer_t;

typedef struct
{
        mach_msg_trailer_type_t     msgh_trailer_type;
        mach_msg_trailer_size_t     msgh_trailer_size;
        mach_port_seqno_t              msgh_seqno;
} mach_msg_seqno_trailer_t;

typedef struct
{
        mach_msg_trailer_type_t    msgh_trailer_type;
        mach_msg_trailer_size_t    msgh_trailer_size;
        mach_port_seqno_t             msgh_seqno;
        security_token_t             msgh_sender;
} mach_msg_security_trailer_t;

typedef struct
{
        mach_msg_trailer_type_t    msgh_trailer_type;
        mach_msg_trailer_size_t    msgh_trailer_size;
        mach_port_seqno_t             msgh_seqno;
        security_token_t              msgh_sender;
        unsigned int                dipc_sender_kmsg;
} mach_msg_dipc_trailer_t;
```

[Figure 72]: Components of a Mach message

- The picture above does not show, but for complex messages, the **mach_msg_body_t** structure, which holds the **msgh_descriptor_count** field, is followed by one or more descriptors, being the most basic of them used for passing a Mach port right). If it is a simple message then the body is only followed by normal data (untyped). Structures of message descriptors are shown below:

```
typedef struct
{
        void*                           pad1;
        mach_msg_size_t                 pad2;
        unsigned int                    pad3 : 24;
        mach_msg_descriptor_type_t      type : 8;
} mach_msg_type_descriptor_t;

typedef struct
{
        mach_port_t                     name;
        mach_msg_size_t                 pad1;
        unsigned int                    pad2 : 16;
        mach_msg_type_name_t        disposition : 8;
        mach_msg_descriptor_type_t      type : 8;
} mach_msg_port_descriptor_t;

typedef struct
{
        void*                       address;
        mach_msg_size_t                size;
        boolean_t                  deallocate : 8;
        mach_msg_copy_options_t         copy : 8;
        unsigned int                    pad1 : 8;
        mach_msg_descriptor_type_t      type : 8;
} mach_msg_ool_descriptor_t;

typedef struct
{
        void*                         address;
        mach_msg_size_t                 count;
        boolean_t                    deallocate : 8;
        mach_msg_copy_options_t           copy : 8;
        mach_msg_type_name_t        disposition : 8;
        mach_msg_descriptor_type_t        type : 8;
} mach_msg_ool_ports_descriptor_t;

typedef union
{
        mach_msg_port_descriptor_t              port;
        mach_msg_ool_descriptor_t       out_of_line;
        mach_msg_ool_ports_descriptor_t   ool_ports;
        mach_msg_type_descriptor_t              type;
} mach_msg_descriptor_t;
```

[Figure 73]: Distinct types of message descriptors for a Mach message

- **Ports** are created by varied reasons and situations. For example, **when a task is created a port is allocated as well as when launchd starts**. On the other hand, **a port is not inherited whether invoking a fork function**.
- If you pay attention to the other **Mach message descriptors**, one of them is **mach_msg_ool_descriptor_t**, which refers to messages that carry the address of a memory region because sender and received shares the memory region. This kind of message transfer is known as **out-of-line transfer (OOL transfer)**.
- The message structure composition, mainly complex messages, are really important while searching for vulnerabilities and, as readers can realize, several fields are really structures whose respective description and meaning relevant.

- **RPC (Remote Procedure Call)** is another mechanism that allows programs to communicate with each other. However, different from other operating systems, macOS offers **the MIG (Mach Interface Generator)**, which produces all necessary code for client-server communication based on specification file (**.defs extension**). Thus, the generated **MIG server** is actually a task which makes available services for clients. Additionally, MIG is also used to implement Mach system calls and Mach interfaces for kernel objects are implemented using MIG via **mig_subsystem** structure:

```
121: typedef struct mig_subsystem {
122:     mig_server_routine_t server;            /* pointer to demux routine */
123:     mach_msg_id_t            start;                /* Min routine number       */
124:     mach_msg_id_t            end;                  /* Max routine number + 1   */
125:     mach_msg_size_t          maxsize;              /* Max reply message size   */
126:     vm_address_t             reserved;             /* reserved for MIG use     */
127:     mig_routine_descriptor
128:         routine[1];                                        /* Routine descriptor array */
129: } *mig_subsystem_t;
130:
131: #ifdef XNU_KERNEL_PRIVATE
132: /* KernelServer MIG routine/subsystem types */
133: typedef void    (*mig_stub_kern_routine_t) (mach_msg_header_t *InHeadP, void *InDataP,
134:     mach_msg_max_trailer_t *InTrailerP, mach_msg_header_t *OutHeadP, void *OutDataP);
135:
136: typedef mig_stub_kern_routine_t mig_kern_routine_t;
137:
138: typedef mig_kern_routine_t (*mig_kern_server_routine_t) (mach_msg_header_t *InHeadP);
139:
140: struct kern_routine_descriptor {
141:     mig_impl_routine_t      impl_routine;      /* Server work func pointer   */
142:     mig_stub_kern_routine_t kstub_routine;     /* Unmarshalling func pointer */
143:     unsigned int            argc;              /* Number of argument words   */
144:     unsigned int            descr_count;       /* Number complex descriptors */
145:     unsigned int            reply_descr_count; /* Number descriptors in reply */
146:     unsigned int            max_reply_msg;     /* Max size for reply msg */
147: };
148:
149: typedef struct kern_routine_descriptor mig_kern_routine_descriptor;
150: typedef struct kern_routine_descriptor *kern_routine_descriptor_t;
151:
152: typedef struct mig_kern_subsystem {
153:     mig_kern_server_routine_t     kserver;     /* pointer to kernel demux routine */
154:     mach_msg_id_t            start;            /* Min routine number */
155:     mach_msg_id_t            end;              /* Max routine number + 1 */
156:     mach_msg_size_t          maxsize;          /* Max reply message size */
157:     vm_address_t             reserved;         /* reserved for MIG use */
158:     mig_kern_routine_descriptor
159:         kroutine[1];                           /* Kernel routine descriptor array */
160: } *mig_kern_subsystem_t;
161: #endif /* XNU_KERNEL_PRIVATE */
162:
```

[Figure 74]: mig_subsystem structure and kernel_routine_descriptor

- There is a special type of Mach port that is an **exception port**, which receives information about the generated exception such as the responsible thread for the exception, the task that contains the thread, and also the exception type, being the **there is a port for exception type for thread, task and host** (from most specific port up to the most general one). As it is anticipated, there is an exception handler associated with an exception thread.
- The **osfmk\mach\exception_types.h** offers a series of information and definition on exceptions such as **exported types, Mach exceptions** and **exception behaviors**:

```
64: /*
65:  *  Machine-independent exception definitions.
66:  */
67:
68: #define EXC_BAD_ACCESS          1       /* Could not access memory */
69: /* Code contains kern_return_t describing error. */
70: /* Subcode contains bad memory address. */
71:
72: #define EXC_BAD_INSTRUCTION     2       /* Instruction failed */
73: /* Illegal or undefined instruction or operand */
74:
75: #define EXC_ARITHMETIC          3       /* Arithmetic exception */
76: /* Exact nature of exception is in code field */
77:
78: #define EXC_EMULATION           4       /* Emulation instruction */
79: /* Emulation support instruction encountered */
80: /* Details in code and subcode fields   */
81:
82: #define EXC_SOFTWARE            5       /* Software generated exception */
83: /* Exact exception is in code field. */
84: /* Codes 0 - 0xFFFF reserved to hardware */
85: /* Codes 0x10000 - 0x1FFFF reserved for OS emulation (Unix) */
86:
87: #define EXC_BREAKPOINT          6       /* Trace, breakpoint, etc. */
88: /* Details in code field. */
89:
90: #define EXC_SYSCALL             7       /* System calls. */
91:
92: #define EXC_MACH_SYSCALL        8       /* Mach system calls. */
93:
94: #define EXC_RPC_ALERT           9       /* RPC alert */
95:
96: #define EXC_CRASH               10      /* Abnormal process exit */
97:
98: #define EXC_RESOURCE            11      /* Hit resource consumption limit */
99: /* Exact resource is in code field. */
100:
101: #define EXC_GUARD               12      /* Violated guarded resource protections */
102:
103: #define EXC_CORPSE_NOTIFY       13      /* Abnormal process exited to corpse state */
```

[Figure 75]: Mach Exceptions

```
106: /*
107:  *  Machine-independent exception behaviors
108:  */
109:
110: #define EXCEPTION_DEFAULT               1
111: /*  Send a catch_exception_raise message including the identity.
112:  */
113:
114: #define EXCEPTION_STATE                 2
115: /*  Send a catch_exception_raise_state message including the
116:  *  thread state.
117:  */
118:
119: #define EXCEPTION_STATE_IDENTITY        3
120: /*  Send a catch_exception_raise_state_identity message including
121:  *  the thread identity and state.
122:  */
123:
124: #define EXCEPTION_IDENTITY_PROTECTED    4
125: /*  Send a catch_exception_raise_identity_protected message including protected task
126:  *  and thread identity.
127:  */
128:
129: #define EXCEPTION_STATE_IDENTITY_PROTECTED   5
130: /*  Send a catch_exception_raise_state_identity_protected message including protected task
131:  *  and thread identity plus the thread state.
132:  */
133:
134: #define MACH_EXCEPTION_BACKTRACE_PREFERRED   0x20000000
135: /* Prefer sending a catch_exception_raise_backtrace message, if applicable */
136:
137: #define MACH_EXCEPTION_ERRORS           0x40000000
138: /*  include additional exception specific errors, not used yet.  */
139:
140: #define MACH_EXCEPTION_CODES            0x80000000
141: /*  Send 64-bit code and subcode in the exception header */
142:
143: #define MACH_EXCEPTION_MASK             (MACH_EXCEPTION_CODES | \
144:                                          MACH_EXCEPTION_ERRORS | \
145:                                          MACH_EXCEPTION_BACKTRACE_PREFERRED)
```

[Figure 76]: Expected Mach Behaviors

- macOS as diverse versions of Linux, offers the possibility for working with **pipes**, which are unidirectional data-flow by nature and are suitable for related processes. A pipe can be unnamed or named, which is also referred to as **fifo**, and allows regular usage with functions associated with file system operations using blocking or non-blocking mode.
- **Apple Even**t is another IPC mechanism created for transporting and releasing messages and as stated previously, it is strongly based on AppleScript.
- **Distributed Objects** (also referred to as D.O.) is provided by Object-C runtime and allows applications to call remote objects from other applications (running locally or remotely) or even the same one. Of course, the impression offers by D.O. is that the remote object is located in the same system similar to modern technologies, frameworks, and applications. D.O. uses methods from **NSConnection class**.
- **Core Foundation** (as well-known as CF framework) offers a range of functions that are responsible for sending and receiving notification, which is composed of a name, an object identifier and a dictionary (probably targeted for an attacker) that effectively carries the information, and such actions occur in the communication of local and even remote processes. I have already touched on this subject when I quickly explained about the local and distributed notification center, which is able to send named messages to all subscribers. The **NSXPCConnection** class, also mentioned previously, is an evolution of Distributed Objects.

On recent versions of macOS, most of code executions happen inside of a sandbox environment, which part of the application (including XPC programs) run inside the sandbox and the other part of the same application runs out of the sandbox, and both communicate to each other through IPC. Moreover, macOS adopts the same approach for system daemons and there are many additional protections that we will review in the next section.

08. Security

When discussing on macOS/iOS security, there are software and hardware contexts, and each one has a really extensive list of topics to be detailed. You already know that exploiting vulnerabilities on Apple devices can be a challenging task due to multiple protections and constant improvements that have been implemented over time, and such evolution occurred in the system security, service security, hardware security, application security and mainly as data are stored and encrypted. About the most important protections themselves I will describe them later, and first we need to discuss the main mechanisms used by macOS and iOS. The first clear fact is that it is harder to exploit a mobile device (for example, iPhone) than only the operating system because the access is different, the involved hardware is also different and even protections on iOS are tighter than macOS. Apple offers a few pages related to security research and also its bug-bounty program, and readers should visit them:

- **Blog:** https://security.apple.com/blog/
- **Security Bounty:** https://security.apple.com/bounty/
- **Bounty Categories:** https://security.apple.com/bounty/categories/

If readers want to learn more about past vulnerabilities then existing CVE can be important:

Apple : Vulnerability Statistics

Products (214) Vulnerabilities (7608) Search products CVSS Report Metasploit Modules

Vulnerability Trends Over Time

Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validation
2015	297	306	0	3	5	0	1	4	0	0	53
2016	166	180	0	5	0	0	0	1	0	1	22
2017	297	295	0	16	0	1	0	0	0	1	57
2018	68	67	0	2	1	0	0	0	0	1	25
2019	111	233	2	17	1	1	0	0	0	1	76
2020	32	152	0	9	5	0	0	0	0	0	42
2021	37	145	0	7	3	0	0	0	1	2	10
2022	50	139	0	3	0	0	1	1	0	0	8
2023	37	58	0	1	0	1	0	0	0	0	1
2024	12	21	0	3	0	0	0	0	0	0	0
2025	1	0	0	0	0	0	0	0	0	0	0
Total	1108	1596	2	66	15	3	2	6	1	6	294

Vulnerabilities by impact types

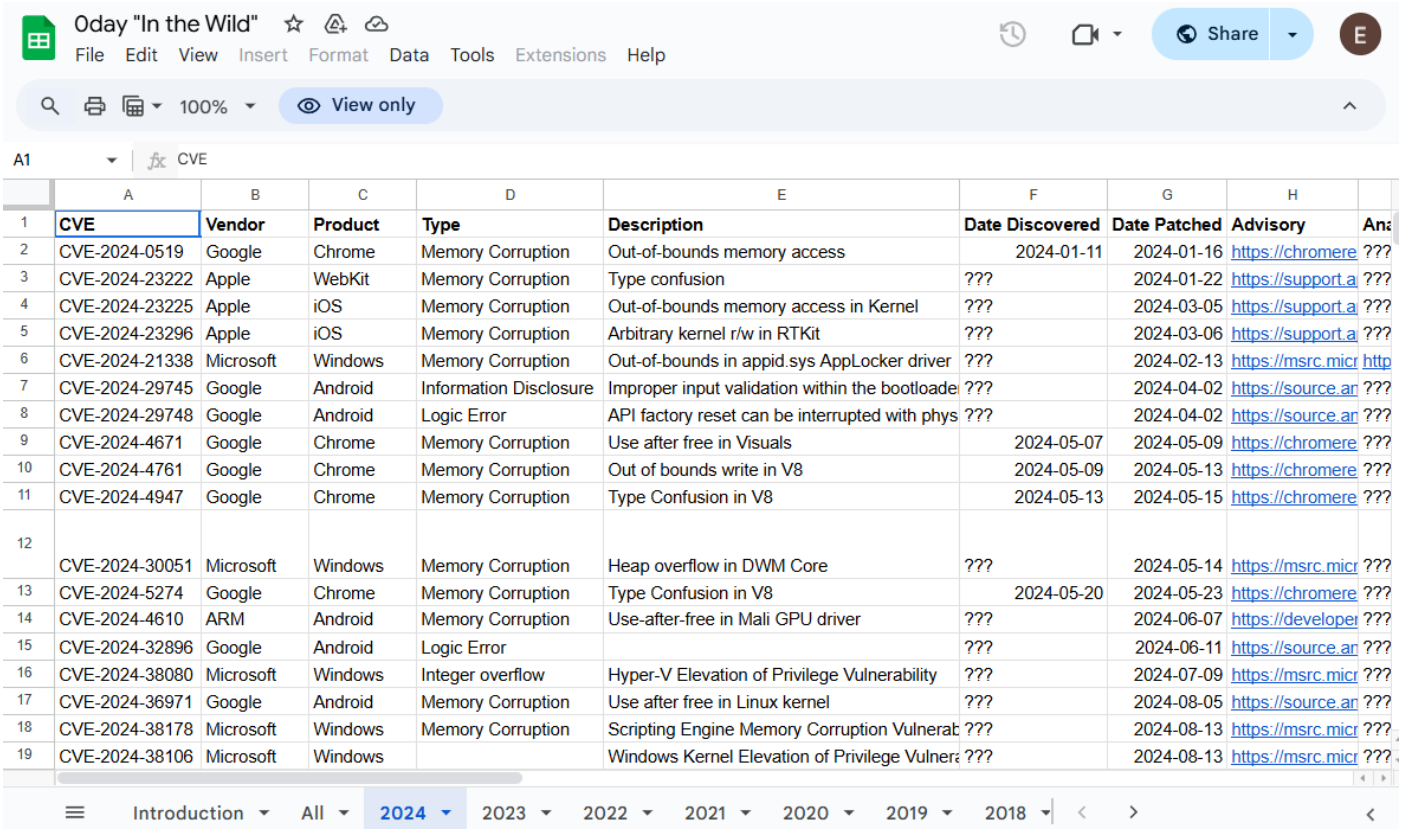
Year	Code Execution	Bypass	Privilege Escalation	Denial of Service	Information Leak
2015	316	35	35	369	94
2016	157	13	13	215	51
2017	313	4	4	362	81
2018	86	0	1	78	26
2019	37	7	6	19	26
2020	80	0	0	27	13
2021	81	8	8	30	8
2022	98	3	3	20	5
2023	59	1	2	22	2
2024	31	4	1	22	5
2025	0	0	0	0	0
Total	1258	75	73	1164	311

[Figure 77]: Existing Apple CVE Statistics | Credits: https://www.cvedetails.com/vendor/49/Apple.html

You can check for details of each vulnerability on https://www.cvedetails.com/vulnerability-list/vendor_id-49/Apple.html. Another excellent source for keeping update about vulnerabilities (not only macOS/iOS) and zero-days is the following spreadsheet made available by Google Project Zero:

- https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/edit?gid=694054923#gid=694054923.

Make sure to check different sheets, as shown below:



0day "In the Wild"

File Edit View Insert Format Data Tools Extensions Help

Share

100% View only

A1 CVE

	A	B	C	D	E	F	G	H	
1	CVE	Vendor	Product	Type	Description	Date Discovered	Date Patched	Advisory	An
2	CVE-2024-0519	Google	Chrome	Memory Corruption	Out-of-bounds memory access	2024-01-11	2024-01-16	https://chromere	???
3	CVE-2024-23222	Apple	WebKit	Memory Corruption	Type confusion	???	2024-01-22	https://support.a	???
4	CVE-2024-23225	Apple	iOS	Memory Corruption	Out-of-bounds memory access in Kernel	???	2024-03-05	https://support.a	???
5	CVE-2024-23296	Apple	iOS	Memory Corruption	Arbitrary kernel r/w in RTKit	???	2024-03-06	https://support.a	???
6	CVE-2024-21338	Microsoft	Windows	Memory Corruption	Out-of-bounds in appid.sys AppLocker driver	???	2024-02-13	https://msrc.micr	http
7	CVE-2024-29745	Google	Android	Information Disclosure	Improper input validation within the bootloade	???	2024-04-02	https://source.an	???
8	CVE-2024-29748	Google	Android	Logic Error	API factory reset can be interrupted with phys	???	2024-04-02	https://source.an	???
9	CVE-2024-4671	Google	Chrome	Memory Corruption	Use after free in Visuals	2024-05-07	2024-05-09	https://chromere	???
10	CVE-2024-4761	Google	Chrome	Memory Corruption	Out of bounds write in V8	2024-05-09	2024-05-13	https://chromere	???
11	CVE-2024-4947	Google	Chrome	Memory Corruption	Type Confusion in V8	2024-05-13	2024-05-15	https://chromere	???
12	CVE-2024-30051	Microsoft	Windows	Memory Corruption	Heap overflow in DWM Core	???	2024-05-14	https://msrc.micr	???
13	CVE-2024-5274	Google	Chrome	Memory Corruption	Type Confusion in V8	2024-05-20	2024-05-23	https://chromere	???
14	CVE-2024-4610	ARM	Android	Memory Corruption	Use-after-free in Mali GPU driver	???	2024-06-07	https://developer	???
15	CVE-2024-32896	Google	Android	Logic Error		???	2024-06-11	https://source.an	???
16	CVE-2024-38080	Microsoft	Windows	Integer overflow	Hyper-V Elevation of Privilege Vulnerability	???	2024-07-09	https://msrc.micr	???
17	CVE-2024-36971	Google	Android	Memory Corruption	Use after free in Linux kernel	???	2024-08-05	https://source.an	???
18	CVE-2024-38178	Microsoft	Windows	Memory Corruption	Scripting Engine Memory Corruption Vulnerab	???	2024-08-13	https://msrc.micr	???
19	CVE-2024-38106	Microsoft	Windows		Windows Kernel Elevation of Privilege Vulnera	???	2024-08-13	https://msrc.micr	???

Introduction All 2024 2023 2022 2021 2020 2019 2018

[Figure 78]: List of CVEs and possibly zero days

Probably you will need to jailbreak an iPhone, and one of the best website to learn what versions and devices the jailbreak is possible is: https://idevicecentral.com/jailbreak-tools/ios-jailbreak-tools/ or even an automatic finder: https://idevicecentral.com/ios-jailbreak-tool-finder/:

You can use our Jailbreak Finder Tool to check available jailbreak tools for your device! You can also find the jailbreak development timeline for iOS 16.0 – 16.5 A12+ here.

On iPhone XR and Newer (A12+):

- **iOS 18.0 – 18.2** is not jailbreakable yet. Several tools like MisakaX, Nugget, etc. may help with iOS customization.
- **iOS 17.0** – **iOS 17.7.2** is not jailbreakable yet. Several tools like MisakaX, Nugget, etc. may help with iOS customization.
- **iOS 16.7** is not jailbreakable yet.
- **iOS 16.5.1 – 16.6.1 can be jailbroken on All Devices using NathanLR Semi-Jailbreak**
- **iOS 15.0 -> iOS 16.5 / 16.5.1** you can jailbreak with **Dopamine 2.x Jailbreak** (**RECOMMENDED**), or Serotonin Jailbreak + TrollStore 2 + RootHide Bootstrap.
- **iOS 15.0 – 15.4.1** is also jailbreakable with **XinaA15 jailbreak** by @xina520. You can also jailbreak with XinaAHere are all compatible Rootless Tweaks!

On iPhone X and older (A11 and below):

- **iPadOS 15.0** – **17.6.1 can be jailbroken with PaleRa1n.**
- **iOS 15.0 – 16.6.1** can be jailbroken with Dopamine 2.0 Jailbreak.
- **iOS 15.0 – 15.8** can also be jailbroken with **nekoJailbreak** and **meowbrek2** which allow you to re-enable palera1n jailbreak without using a computer or PaleRa1n app after initially jailbreaking with it.

[Figure 79]: iDevice Central | Jailbreak Tools for All iOS Versions

A suggestion is to retrieve a list of known macOS, iOS and related vulnerabilities of the last years to learn about them. Using the excellent **CVEProject/cvelistV5 project**, which is updated many times during the day (you need to update it using **git pull** command):

- **git clone** https://github.com/CVEProject/cvelistV5
- **cd cvelistV5/cves**

A fair note is that they are still migrating old records and fixing eventual issues, so problems, formatting and even lacking records might happen.

The following two scripts search for CVEs proposed by Apple and related to a pattern (macOS, iOS) and, once you pick up a CVE, the second script tries to format it (it is imprecise because records do not follow a fixed format that always have the same fields). I saved both scripts in **cvelistV5/cves** directory**:**

```python
import re
import os
import sys

# Check for the correct number of arguments
if len(sys.argv) != 3:
    print("Usage: python search_values.py <pattern> <directory>")
    sys.exit(1)

pattern_input = sys.argv[1]
directory = sys.argv[2]

pattern = re.compile(rf'"(title|value)": ".*{pattern_input}.*"', re.IGNORECASE)
assigner_pattern = re.compile(r'"assignerShortName": "apple"', re.IGNORECASE)

def extract_value(content):
    match = re.search(r'"value": "(.*?)"', content, re.IGNORECASE)
    if match:
        return match.group(1)
    return None

# Traverse the specified directory
for root, _, files in os.walk(directory):
    for file in files:
        if file.endswith('.json'):
            filepath = os.path.join(root, file)
            with open(filepath, 'r', encoding='utf-8') as f:
                content = f.read()
                if pattern.search(content) and assigner_pattern.search(content):
                    value = extract_value(content)
                    if value:
                        relative_path = os.path.relpath(filepath, start=directory)
                        print(f"\n[+] {relative_path}:\t{value}")
```

[Figure 80]: search_cve.py: searches for CVEs related to a pattern in a provided directory

Although it seems to be important to comment right now, I have fixed the assigner to Apple because there are hundreds of other vulnerabilities related to Apple's products, but that are not from Apple products themselves. Moreover, I have also focused on the complete description because it is more informative.

```
root@ubuntu01:~/github/cvelistV5/cves# ls
1999  2002  2005  2008  2011  2014  2017  2020  2023  delta.json     scripts
2000  2003  2006  2009  2012  2015  2018  2021  2024  deltaLog.json  search_cve.py
2001  2004  2007  2010  2013  2016  2019  2022  2025  parse_cve.py
root@ubuntu01:~/github/cvelistV5/cves#
root@ubuntu01:~/github/cvelistV5/cves# python search_cve.py macOS 2024

[+] 40xxx/CVE-2024-40779.json:  An out-of-bounds read was addressed with improved bounds c
hecking. This issue is fixed in iOS 16.7.9 and iPadOS 16.7.9, Safari 17.6, iOS 17.6 and iP
adOS 17.6, watchOS 10.6, tvOS 17.6, visionOS 1.3, macOS Sonoma 14.6. Processing maliciousl
y crafted web content may lead to an unexpected process crash.

[+] 40xxx/CVE-2024-40842.json:  An issue was addressed with improved validation of environ
ment variables. This issue is fixed in macOS Sequoia 15. An app may be able to access user
-sensitive data.

[+] 40xxx/CVE-2024-40799.json:  An out-of-bounds read issue was addressed with improved in
put validation. This issue is fixed in iOS 16.7.9 and iPadOS 16.7.9, macOS Ventura 13.6.8,
 macOS Monterey 12.7.6, iOS 17.6 and iPadOS 17.6, watchOS 10.6, tvOS 17.6, visionOS 1.3, m
acOS Sonoma 14.6. Processing a maliciously crafted file may lead to unexpected app termina
tion.

[+] 40xxx/CVE-2024-40816.json:  An out-of-bounds read was addressed with improved input va
lidation. This issue is fixed in macOS Sonoma 14.6, macOS Monterey 12.7.6, macOS Ventura 1
3.6.8. A local attacker may be able to cause unexpected system shutdown.

[+] 40xxx/CVE-2024-40826.json:  A privacy issue was addressed with improved handling of fi
les. This issue is fixed in iOS 18 and iPadOS 18, macOS Sequoia 15. An unencrypted documen
t may be written to a temporary file when using print preview.

[+] 40xxx/CVE-2024-40837.json:  A permissions issue was addressed with additional restrict
ions. This issue is fixed in macOS Sequoia 15. An app may be able to access protected user
 data.

[+] 40xxx/CVE-2024-40788.json:  A type confusion issue was addressed with improved memory
handling. This issue is fixed in iOS 16.7.9 and iPadOS 16.7.9, macOS Ventura 13.6.8, macOS
 Monterey 12.7.6, iOS 17.6 and iPadOS 17.6, watchOS 10.6, tvOS 17.6, visionOS 1.3, macOS S
onoma 14.6. A local attacker may be able to cause unexpected system shutdown.

[+] 40xxx/CVE-2024-40805.json:  A permissions issue was addressed with additional restrict
ions. This issue is fixed in watchOS 10.6, macOS Sonoma 14.6, iOS 17.6 and iPadOS 17.6, tv
OS 17.6. An app may be able to bypass Privacy preferences.

[+] 40xxx/CVE-2024-40794.json:  This issue was addressed through improved state management
. This issue is fixed in macOS Sonoma 14.6, iOS 17.6 and iPadOS 17.6, Safari 17.6. Private
 Browsing tabs may be accessed without authentication.
```

[Figure 81]: search_cve.py's output (truncated)

Each line represents a reported vulnerability, and for this reason the script shows the name of the JSON file and its respective complete description. I have not used the title of the report because it is not descriptive in most of the opportunities, and it would not be useful.

As I stated previously, the second script tries to format the provided JSON file and retrieve a few relevant fields. You can adapt it accordingly your needs. As reports can bring varied amounts of details and not all of them follow a fixed structure, you must not expect that the script to work with all provided JSON files and, for sure, adjustments will be necessary:

```python
import json
import argparse
import sys

# Set up argument parser
parser = argparse.ArgumentParser(description='Parse specific fields from a JSON file.')
parser.add_argument('file', type=str, nargs='?', help='Path to the JSON file')

# Parse arguments
args = parser.parse_args()

# Check if the file argument is provided
if not args.file:
    print("Usage: python3 parse_json.py <path_to_json_file>")
    sys.exit(1)

# Read JSON data from file
with open(args.file, 'r') as file:
    json_data = file.read()

# Parse JSON
data = json.loads(json_data)

# Extract required fields with checks
url = data.get('containers', {}).get('cna', {}).get('references', [{}])[0].get('url', 'N/A')
date_published = data.get('cveMetadata', {}).get('datePublished', 'N/A')

# Extract all lessThan versions, descriptions, and products with checks
less_than_list = [version['lessThan'] for affected in data.get('containers', {}).get('cna', {}).get('affected', []) for version in affected.get('versions', []) if 'lessThan' in version]
cwe_list = [desc.get('cweId', 'N/A') for problem in data.get('containers', {}).get('adp', [{}])[0].get('problemTypes', []) for desc in problem.get('descriptions', []) if 'cweId' in desc]
description_list = [desc.get('description', 'N/A') for problem in data.get('containers', {}).get('cna', {}).get('problemTypes', []) for desc in problem.get('descriptions', [])]
product_list = [affected.get('product', 'N/A') for affected in data.get('containers', {}).get('cna', {}).get('affected', [])]

# Extract Base Score, User Interaction, Base Severity, and Privileges Required under metrics -> cvssV3_1
base_score = 'N/A'
user_interaction = 'N/A'
base_severity = 'N/A'
privileges_required = 'N/A'
attack_vector = 'N/A'
attack_complexity = 'N/A'
for adp in data.get('containers', {}).get('adp', []):
    for metric in adp.get('metrics', []):
        if 'cvssV3_1' in metric:
            cvss_v3 = metric['cvssV3_1']
            base_score = cvss_v3.get('baseScore', 'N/A')
            user_interaction = cvss_v3.get('userInteraction', 'N/A')
            base_severity = cvss_v3.get('baseSeverity', 'N/A')
            privileges_required = cvss_v3.get('privilegesRequired', 'N/A')
            attack_vector = cvss_v3.get('attackVector', 'N/A')
            attack_complexity = cvss_v3.get('attackComplexity', 'N/A')
            break

# Extract Technical Impact and values
value_list = []
for description in data.get('containers', {}).get('cna', {}).get('descriptions', []):
    if 'value' in description:
        value_list.append(description['value'])

```

```
59 # Print extracted fields with specified format
60 print(f"[+] url: {url}")
61 print(f"[+] datePublished: {date_published}")
62 print(f"[+] lessThan versions:")
63 for version in less_than_list:
64     print(f"  - {version}")
65 print(f"[+] cwe:")
66 for cwe in cwe_list:
67     print(f"  - {cwe}")
68 print(f"[+] descriptions:")
69 for description in description_list:
70     print(f"  - {description}")
71 print(f"[+] products:")
72 for product in product_list:
73     print(f"  - {product}")
74 print(f"[+] baseScore: {base_score}")
75 print(f"[+] userInteraction: {user_interaction}")
76 print(f"[+] baseSeverity: {base_severity}")
77 print(f"[+] attackVector: {attack_vector}")
78 print(f"[+] attackComplexity: {attack_complexity}")
79 print(f"[+] values:")
80 for value in value_list:
81     print(f"  - {value}")
```

[Figure 82]: parse_cve.py: it parses a provided JSON file

An example of output is the following one:

```
root@ubuntu01:~/github/cvelistV5/cves# python parse_cve.py 2024/54xxx/CVE-2024-54465.json

[+] url: https://support.apple.com/en-us/121839
[+] datePublished: 2024-12-11T22:56:56.616Z
[+] lessThan versions:
  - 15.2
[+] cwe:
  - CWE-281
[+] descriptions:
  - An app may be able to elevate privileges
[+] products:
  - macOS
[+] baseScore: 9.8
[+] userInteraction: NONE
[+] baseSeverity: CRITICAL
[+] attackVector: NETWORK
[+] attackComplexity: LOW
[+] values:
  - A logic issue was addressed with improved state management. This issue is fixed in ma
cOS Sequoia 15.2. An app may be able to elevate privileges.
root@ubuntu01:~/github/cvelistV5/cves#
```

[Figure 83]: parse_cve.py output

The most important fields from the output above url, dataPublished, lessThen version, description, and values. The remaining fields can help and provide a guideline for further investigation. Of course, you can read to the JSON file directly.

The general idea is to make a quick and compact review of only a few security mechanisms involved:

- macOS and iOS implement **MAC (Mandatory Access Control) framework**, which represents its great and powerful foundation. Other security features are built on the **MACF** foundation.

- There are associated MACF policies and respective list of rules, and also there is the **security\mac_policy.h file**, which contains information about kernel interfaces for MACF policy modules and also defines the list of supported operations. Examples of features that need **MACF** to work appropriately are **AMFI (AppleMobileFileIntegrity)** and **Sandbox**.

- In Apple devices, native binaries are digitally signed, and their respective integrity are checked by verifying exactly such a signature to ensure that none has changed and compromised the file's integrity. Additionally, once the binary is signed, it is possible to know who has signed it. An example of usage of **codesign** command, which is used to check and show signatures, is shown below:

```
alexandreborges@Mac ~ % codesign -vvvvd /usr/bin/otool
Executable=/usr/bin/otool
Identifier=com.apple.dt.xcode_select.tool-shim
Format=Mach-O universal (x86_64 arm64e)
CodeDirectory v=20400 size=316 flags=0x0(none) hashes=4+2 location=embedded
Platform identifier=15
VersionPlatform=1
VersionMin=919040
VersionSDK=919040
Hash type=sha256 size=32
CandidateCDHash sha256=09f2c2011000bbded1c03ca23a032c98a8cf6358
CandidateCDHashFull sha256=09f2c2011000bbded1c03ca23a032c98a8cf635802059c86
991005d33a05e2c0
Hash choices=sha256
CMSDigest=09f2c2011000bbded1c03ca23a032c98a8cf635802059c86991005d33a05e2c0
CMSDigestType=2
Executable Segment base=0
Executable Segment limit=4096
Executable Segment flags=0x1
Page size=4096
CDHash=09f2c2011000bbded1c03ca23a032c98a8cf6358
Signature size=4442
Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA
Signed Time=Jun 29, 2024 at 15:57:01
Info.plist entries=17
TeamIdentifier=not set
Sealed Resources=none
Internal requirements count=1 size=84
alexandreborges@Mac ~ %
```

[Figure 84]: codesign command

- There is another model that is **ad-hoc signatures**, whose signatures (actually only hashes and not the digital certificate) are stored into a cache (known as trust cache) and that can be protected by AMFI, which is an exclusive iOS topic.

- Since 2023 Apple introduced the concept of **notarization**, where developers must submit the code to the Apple notary service, which will scan the software for any malicious content and also check the code signature. If everything is ok, so a ticket is generated, and the developer can stamp such a ticket to the software, and the online service will be showing such a ticket, which will be available for the **Gatekeeper**, which is a runtime protection that check downloaded application for potential

infection by malware, and whose effect is applied by **/usr/libexec/syspolicyd** daemon and database is stored in **/var/db/SystemPolicy** SQLite database.

- In the example below, commands are checking for third-party application's signature (010 Editor), the respective **code signature requirements** and its **notarization,** respectively:

```
alexandreborges@Mac ~ % codesign -vd /Applications/010\ Editor.app/Cont
ents/MacOS/010\ Editor
Executable=/Applications/010 Editor.app/Contents/MacOS/010 Editor
Identifier=com.SweetScape.010Editor
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20500 size=96164 flags=0x10000(runtime) hashes=2998+3 l
ocation=embedded
Signature size=9078
Timestamp=May 2, 2023 at 14:43:27
Info.plist entries=11
TeamIdentifier=252VCA66Z8
Runtime Version=10.14.0
Sealed Resources version=2 rules=13 files=45
Internal requirements count=1 size=184
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % codesign -r- -d /Applications/010\ Editor.app/C
ontents/MacOS/010\ Editor
Executable=/Applications/010 Editor.app/Contents/MacOS/010 Editor
designated => identifier "com.SweetScape.010Editor" and anchor apple ge
neric and certificate 1[field.1.2.840.113635.100.6.2.6] /* exists */ an
d certificate leaf[field.1.2.840.113635.100.6.1.13] /* exists */ and ce
rtificate leaf[subject.OU] = "252VCA66Z8"
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % spctl -a -vvv -t install /Applications/010\ Edi
tor.app/Contents/MacOS/010\ Editor
/Applications/010 Editor.app/Contents/MacOS/010 Editor: accepted
source=Notarized Developer ID
origin=Developer ID Application: SweetScape Software Inc. (252VCA66Z8)
alexandreborges@Mac ~ %
```

[Figure 85]: checking signature, policy rules

- macOS and iOS also use **entitlements, which are rights that grant to an executable capability for performing diverse types of operations**, and such entitlements are stored into the code signature as list of properties.

- As **entitlements** represent a security layer, in which **capabilities (rights) are granted to applications**, they establish a way of control on the target application, limiting what the application can or cannot do.

- There are a couple of lists including **entitlements**:

  - **Apple:** https://developer.apple.com/documentation/bundleresources/entitlements
  - **OS X/iOS Entitlement Database:** https://newosxbook.com/ent.php

- An example of command to **list entitlements** of an application follows below:

```
alexandreborges@Mac /Applications % codesign -d --entitlements - /Applications/Microsoft\ Edge.app
Executable=/Applications/Microsoft Edge.app/Contents/MacOS/Microsoft Edge
[Dict]
        [Key] com.apple.application-identifier
        [Value]
                [String] UBF8T346G9.com.microsoft.edgemac
        [Key] com.apple.developer.associated-domains
        [Value]
                [Array]
                        [String] webcredentials:login.microsoft.com
                        [String] webcredentials:login.microsoftonline.us
                        [String] webcredentials:login.partner.microsoftonline.cn
        [Key] com.apple.developer.associated-domains.applinks.read-write
        [Value]
                [Bool] true
        [Key] com.apple.developer.web-browser.public-key-credential
        [Value]
                [Bool] true
        [Key] com.apple.security.application-groups
        [Value]
                [Array]
                        [String] UBF8T346G9.com.microsoft.oneauth
        [Key] com.apple.security.device.audio-input
        [Value]
                [Bool] true
        [Key] com.apple.security.device.bluetooth
        [Value]
                [Bool] true
        [Key] com.apple.security.device.camera
        [Value]
                [Bool] true
        [Key] com.apple.security.device.print
        [Value]
                [Bool] true
        [Key] com.apple.security.device.usb
        [Value]
                [Bool] true
        [Key] com.apple.security.personal-information.location
        [Value]
                [Bool] true
        [Key] com.apple.security.personal-information.photos-library
        [Value]
                [Bool] true
        [Key] keychain-access-groups
        [Value]
                [Array]
                        [String] UBF8T346G9.com.google.common.folsom
                        [String] UBF8T346G9.com.microsoft.edgemac.devicetrust
                        [String] UBF8T346G9.com.microsoft.edgemac.unexportable-keys
                        [String] UBF8T346G9.com.microsoft.edgemac.webauthn
                        [String] UBF8T346G9.com.microsoft.identity.universalstorage
                        [String] UBF8T346G9.com.microsoft.edgemac.webauthn-uvk
alexandreborges@Mac /Applications %
```

[Figure 86]: Listing entitlements of an application

- Previously I mentioned **Gatekeeper**, which is one of the responsible frameworks for protecting macOS against malware threats, and a feature that already existed before **Gatekeeper**, but works and it is used by such a framework protection is the **file quarantine attribute** for any binary downloaded from the Internet. To reveal extended attributes, which quarantine attribute is one of them, run:

```
alexandreborges@Mac Downloads % ls -l@ googlechrome.dmg
-rw-r--r--@ 1 alexandreborges  staff  222144160 Jun  5  2023 googlechrome.dmg
        com.apple.macl         72
        com.apple.metadata:kMDItemDownloadedDate              53
        com.apple.metadata:kMDItemWhereFroms          146
        com.apple.quarantine           57
alexandreborges@Mac Downloads %
alexandreborges@Mac Downloads % ls -l@ 010EditorMac64Installer13.0.2.dmg
-rw-r--r--@ 1 alexandreborges  staff  25349544 Jun 30  2023 010EditorMac64Installer13.0.2.dmg
        com.apple.macl        72
        com.apple.metadata:kMDItemWhereFroms         144
        com.apple.provenance          11
        com.apple.quarantine          21
alexandreborges@Mac Downloads %
```

[Figure 87]: Listing extended attributes

- Readers can also list extended attributes using **xattr -l <binary>** command. If readers want to interact with **Gatekeeper** and its associated database with the list of approved binaries, it is possible to list the content of the policy database by executing:

```
alexandreborges@Mac ~ % sudo spctl --status
assessments enabled
alexandreborges@Mac ~ % sudo spctl --list | head -20
8[Apple System] P20 allow lsopen
        anchor apple
3[Apple System] P20 allow execute
        anchor apple
2[Apple Installer] P20 allow install
        anchor apple generic and certificate 1[subject.CN] = "Apple Software Updat
e Certification Authority"
17[Testflight] P10 allow execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.1] exi
sts and certificate leaf[field.1.2.840.113635.100.6.1.25.1] exists
10[Mac App Store] P10 allow install
        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.10]
 exists
5[Mac App Store] P10 allow install
        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.10]
 exists
4[Mac App Store] P10 allow execute
        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.9]
exists
16[Notarized Developer ID] P5 allow lsopen
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exi
sts and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and notarized
12[Notarized Developer ID] P5 allow install
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exi
sts and (certificate leaf[field.1.2.840.113635.100.6.1.14] or certificate leaf[fie
ld.1.2.840.113635.100.6.1.13]) and notarized
11[Notarized Developer ID] P5 allow execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exi
salexandreborges@Mac ~ %
```

[Figure 88]: Listing the Gatekeeper's policy database

- Of course, accessing the **System Settings → Privacy & Security → Security** is a better and recommended option to check and eventually change Gatekeeper settings.

- Another interesting feature from **Gatekeeper** is **path randomization** (as known as **application translocation**), which when we run a downloaded application, such application is executed from a randomized path to prevent attackers of learning from where such application is launched and replace it by a malicious code. For example, we can check if an application would be translocated or not (**translocate-policy-check**) or it has been translocated or not (**translocate-status-check**) by running:

    - **security translocate-policy-check <application>**
    - **security translocate-policy-check Downloads/010EditorMac64Installer13.0.2.dmg**
    - **security translocate-status-check <application>**
    - **security translocate-status-check Downloads/010EditorMac64Installer13.0.2.dmg**

- Security is enforced in user-mode and kernel-mode, and nowadays it is not allowed to load third-party kernel extensions without explicit user's approval and also restarting the system. Additionally, it is also required to configure the **secure boot** to **Reduced Security** in machines with Apple silicon (ARM processor).

- **AMFI (Apple Mobile File Integrity)** is a mechanism composed of a kernel component (**/System/Library/Extensions/AppleMobileFileIntegrity.kext**) that cannot be unloaded and user-mode component (**/usr/libexec/amfid**), and it is directly involved with security action such as:

    - signature validation, mainly by **amfid** (user-mode).
    - protection against unsigned code injections:
        - applications allocating anonymous memory through **mmap( )** with **MAP_JIT flag** enabled, which **com.apple.security.cs.allow-jit** entitlement, which is required if the application has **Hardened Runtime** capability,
        - only one memory region allocation with this flag is allowed.
        - one of the main concerns of the **AMFI** is on allocation with executable protection.
    - allowing or restricting **debugging** by intercepting ptrace calls (it is not quite different from anti-debugging tricks). Debugging a target process is only possible when **SIP is disabled**, or the **target is unsigned** or even the target has the **get_task_allow** entitlement.
    - accessing task ports on iOS. Note that allowing **send rights** to **task ports** grants unlimited control over such ports.
    - memory protection, when **AMFI** intercepts mprotect calls and check for any attempt of a caller forcing a region to be accepted as trusted and as if it had a valid code signature. In **osfmk\mach\vm_prot.h** file there are this protection and many other ones:

```
146: #define VM_PROT_TRUSTED         ((vm_prot_t) 0x20)
147: #endif /* PRIVATE */
148:
149: /*
150:  *      Another invalid protection value.
151:  *  Indicates that the other protection bits are to be applied as a mask
152:  *  against the actual protection bits of the map entry.
153:  */
```

[Figure 89]: osfmk\mach\vm_prot.h file: VM_PROT_TRUSTED

- No doubt, **AMFI mechanism** is one of many challenges to be overcome during jailbreaking and, over the past, the user-mode component (**amfid**) has been targeted. **AMFI works in tandem with the Sandbox mechanism,** and both use **MACF** as its foundation.

- I have mentioned **Hardened Runtime** because its role is to protect the runtime of software against a few classes of exploits and techniques such as code injection, DLL hijacking and process memory space tampering, and such protection can be enabled on Xcode. As expected, this protection is based on the fact of that **SIP (System Integrity Protection)** is enabled by default.

- Sometimes we want to know the dependencies and dependents of the given kernel extension, and in this specific case, we want to know the dependencies of **com.apple.driver.AppleMobileFileIntegrity** and also dependents on this module. Therefore, I provide the script below (it can be improved a lot) that receives a kernel extension as an argument and provides us both dependencies and dependents list:

```
alexandreborges@alexandremacos amfi_demo % cat list_linked.sh
#!/bin/bash

# Redirect all errors to /dev/null
exec 2>/dev/null

# Check if an argument is provided
if [ -z "$1" ]; then
    echo "Usage: $0 <bundle-identifier>"
    exit 1
fi

# Extract the bundle identifier from the command line argument

bundle_identifier="$1"

# Extract the indices from the `Linked Against` field
linked_indices=$(kmutil showloaded --list-only --bundle-identifier "$bundle_identifier"
| sed -e 's/^.*<\(.*\)>$/\1/' | tr ' ' '\n')

# Loop through the indices and list the corresponding kexts
count=31
printf "\nKernel extensions dependencies:\n"
printf '%*s\n' "$count" | tr ' ' '-'

for index in $linked_indices; do
    kmutil showloaded --list-only | grep -E "^ *$index " | awk '{print $6}'
done

# Extract the index of the provided kernel extension
count=29
printf "\nKernel extensions dependents:\n"
printf '%*s\n' "$count" | tr ' ' '-'
target_index=$(kmutil showloaded --list-only --bundle-identifier "$bundle_identifier" |
awk '{print $1}')
kmutil showloaded --list-only | grep "<.* $target_index .*>" | awk '{print $6}'
printf "\n"
alexandreborges@alexandremacos amfi_demo %
```

[Figure 90]: Script to list dependencies and dependents

- It is time to assess the **script** using **com.apple.driver.AppleMobileFileIntegrity kernel** extension and, if you do not remember this name then it is easier to list all kernel extensions and filter it using a key as "*integrity*", as shown below:

```
alexandreborges@alexandremacos amfi_demo %  kmutil showloaded | grep -i integrity
No variant specified, falling back to release
   18   15 0xfffffe000752de30 0x1b543    0x1b543    com.apple.driver.AppleMobileFileInte
grity (1.0.5) 139E415A-01B3-3728-A2AA-139A4CE81EC3 <17 16 15 8 7 6 5 4 3 2 1>
alexandreborges@alexandremacos amfi_demo %
alexandreborges@alexandremacos amfi_demo % ./list_linked.sh com.apple.driver.AppleMobile
FileIntegrity

Kernel extensions dependencies:
-------------------------------
com.apple.iokit.CoreAnalyticsFamily
com.apple.security.AppleImage4
com.apple.kext.CoreTrust
com.apple.kec.corecrypto
com.apple.kpi.unsupported
com.apple.kpi.private
com.apple.kpi.mach
com.apple.kpi.libkern
com.apple.kpi.iokit
com.apple.kpi.dsep
com.apple.kpi.bsd

Kernel extensions dependents:
-----------------------------
com.apple.security.sandbox
com.apple.AppleSystemPolicy
com.apple.iokit.IOUSBHostFamily
com.apple.driver.AppleUSBTDM
com.apple.driver.AppleSEPKeyStore
com.apple.iokit.EndpointSecurity
com.apple.iokit.IOUserEthernet
com.apple.iokit.IO80211Family
com.apple.driver.AppleBCMWLANCore
com.apple.iokit.IOGPUFamily
com.apple.AGXG14G
com.apple.iokit.IONVMeFamily
com.apple.iokit.IOMobileGraphicsFamily
com.apple.iokit.IOMobileGraphicsFamily-DCP
com.apple.driver.AppleEmbeddedUSBHost

alexandreborges@alexandremacos amfi_demo %
```

[Figure 91]: Output containing dependencies and dependents

- The output above, mainly in the dependents section, shows as other features such sandboxing and **SystemPolicy framework**, as well as a series of other kernel extensions.

- It you want to retrieve the **AMFI kernel component** (**AppleMobileFileIntegrity.kext**) for further reversing and analysis then you need to uncompress the **kernelcache** and then extract the kernel extension. Therefore, the recommended steps are:

    - create a separate folder and go into it: **mkdir amfi_demo ; cd amfi_demo**

- copy the kernel cache into the created folder. The key point is the location of the kernel cache: **cp /System/Volumes/Preboot/*/boot/*/System/Library/Caches/com.apple.kernelcaches/kernelcache .** -- where the '*' is replaced by an id that represents the name of the folder.
- Decompress the kernel cache: **ipsw kernel dec kernelcache**
- Extract the kernel extension: **ipsw kernel extract kernelcache.decompressed com.apple.driver.AppleMobileFileIntegrity --output amfi_kernel_extension.**

```
alexandreborges@alexandremacos ~ % mkdir amfi_demo
alexandreborges@alexandremacos ~ % cd amfi_demo
alexandreborges@alexandremacos amfi_demo % cp /System/Volumes/Preboot/A67AA0EF-7DA
7-4705-AFEC-D38E38921D65/boot/FB580854E5C13E59B0BDA7F42EB463B0E719B1ABA37FF986A2FA
A9FC8DA8878CFA7D83C0137DDABA6FD2F94C24DB545B/System/Library/Caches/com.apple.kerne
lcaches/kernelcache .
alexandreborges@alexandremacos amfi_demo % ls -lh kernelcache
-rw-r--r--  1 alexandreborges  staff    28M Jan 14 01:30 kernelcache
alexandreborges@alexandremacos amfi_demo %

alexandreborges@alexandremacos amfi_demo % ipsw kernel dec kernelcache
   • Decompressing KernelManagement kernelcache
      • Created kernelcache.decompressed
alexandreborges@alexandremacos amfi_demo % ls -lh
total 272608
-rw-r--r--  1 alexandreborges  staff    28M Jan 14 01:30 kernelcache
-rw-r-----  1 alexandreborges  staff   105M Jan 14 01:34 kernelcache.decompressed
alexandreborges@alexandremacos amfi_demo % ipsw kernel extract kernelcache.decompr
essed com.apple.driver.AppleMobileFileIntegrity --output amfi_kernel_extension
   • Created amfi_kernel_extension/com.apple.driver.AppleMobileFileIntegrity
alexandreborges@alexandremacos amfi_demo %
alexandreborges@alexandremacos amfi_demo % ls -lh
total 272608
drwxr-xr-x  3 alexandreborges  staff    96B Jan 14 01:36 amfi_kernel_extension
-rw-r--r--  1 alexandreborges  staff    28M Jan 14 01:30 kernelcache
-rw-r-----  1 alexandreborges  staff   105M Jan 14 01:34 kernelcache.decompressed
alexandreborges@alexandremacos amfi_demo %
alexandreborges@alexandremacos amfi_demo % ls -lh amfi_kernel_extension/com.apple.
driver.AppleMobileFileIntegrity
-rwxr-xr-x  1 alexandreborges  staff   2.0M Jan 14 01:36 amfi_kernel_extension/com
.apple.driver.AppleMobileFileIntegrity
alexandreborges@alexandremacos amfi_demo %
alexandreborges@alexandremacos amfi_demo % file amfi_kernel_extension/com.apple.dr
iver.AppleMobileFileIntegrity
amfi_kernel_extension/com.apple.driver.AppleMobileFileIntegrity: Mach-O 64-bit kex
t bundle arm64e
alexandreborges@alexandremacos amfi_demo %
```

[Figure 92]: Extracting AppleMobileFileIntegrity.kext extension

- As readers have already realized, mechanisms presented so far such as **Code Signing** (only trusted programs are authorized to run), and **Gatekeeper** (only verified developers and code signed applications can run), for example are implemented as **MACF** policies.

- In general, the purpose of **MACF** is to intercept the communication between user-mode and kernel-mode, which includes system calls and kernel events, and based on present MAC Policy Modules, a

decision (allow or deny) is taken. Moreover, there is a series of extended interfaces that can interact with **MACF** via user-mode or kernel mode. Check the **security\mac.h file**:

```
/*
 * Extended non-POSIX.1e interfaces that offer additional services
 * available from the userland and kernel MAC frameworks.
 */
#ifdef __APPLE_API_PRIVATE
__BEGIN_DECLS
int      __mac_execve(char *fname, char **argv, char **envv, mac_t _label);
int      __mac_get_fd(int _fd, mac_t _label);
int      __mac_get_file(const char *_path, mac_t _label);
int      __mac_get_link(const char *_path, mac_t _label);
int      __mac_get_pid(pid_t _pid, mac_t _label);
int      __mac_get_proc(mac_t _label);
int      __mac_set_fd(int _fildes, const mac_t _label);
int      __mac_set_file(const char *_path, mac_t _label);
int      __mac_set_link(const char *_path, mac_t _label);
int      __mac_mount(const char *type, const char *path, int flags, void *data,
    struct mac *label);
int      __mac_get_mount(const char *path, struct mac *label);
int      __mac_set_proc(const mac_t _label);
int      __mac_syscall(const char *_policyname, int _call, void *_arg);
__END_DECLS
#endif /*  APPLE API PRIVATE*/
```

[Figure 93]: security\mac.h file (truncated)

- Communication with **MAC Framework** can occurs through several mechanisms such as extended system calls, vfs, sockets and pipes. The **MACF** files **as mac_framework.h, mac_label.c mac_priv.c, mac_socket.c, mac_vfs.c** and many other ones can be found in **security/** directory from the **XNU** source code. For example, the **__mac_syscall routine** (from **security\mac_base.c**) is shown below:

```
/*
 * __mac_syscall: Perform a MAC policy system call
 *
 * Parameters:    p                       Process calling this routine
 *                uap                     User argument descriptor (see below)
 *                retv                    (Unused)
 *
 * Indirect:      uap->policy             Name of target MAC policy
 *                uap->call               MAC policy-specific system call to perform
 *                uap->arg                MAC policy-specific system call arguments
 *
 * Returns:        0                      Success
 *                !0                      Not success
 *
 */
int
__mac_syscall(proc_t p, struct __mac_syscall_args *uap, int *retv __unused)
{
    struct mac_policy_conf *mpc;
    char target[MAC_MAX_POLICY_NAME];
    int error;
    u_int i;
    size_t ulen;

    error = copyinstr(uap->policy, target, sizeof(target), &ulen);
    if (error) {
        return error;
    }
    AUDIT_ARG(value32, uap->call);
    AUDIT_ARG(mac_string, target);

    error = mac_proc_check_mac_syscall(p, target, uap->call);
    if (error) {
        return error;
    }
```

[Figure 94]: __mac_syscall from security\mac_base.c (truncated)

- There are other frameworks and mechanisms that are also implemented as MACF policies such as **Sandboxing** (applications are run in a restricted context), **System Integrity Protection** (SIP -- it is a system level control that protects critical system files and even directories) and **TCC Framework** (establishes access control to sensitive information, files and directories).

- Unfortunately, not all mechanisms are open-source, and for example, **Sandboxing** is not included in the open-source projects made available by Apple. Furthermore, the implementation and control offered to the user is different between macOS and iOS.

- On macOS, applications are isolated within containers, which are automatically created in **Library/Containers** folder under each home directory. Inside this location there are a series of identifiers (named **CFBundleIdentifier**), which represent each application:

```
alexandreborges@Mac Containers % ls -lh | head -20
total 0
drwx------@ 4 alexandreborges  staff   128B Jul 29 02:31 3303CF40-DF0E-4DF2-96EF-5D6CEC74BE48
drwx------@ 4 alexandreborges  staff   128B Jul 29 02:25 D9114BE9-AB44-4007-8CE4-64E5A15AED0A
drwx------@ 4 alexandreborges  staff   128B Aug 24 20:48 com.apple.AMPArtworkAgent
drwx------@ 4 alexandreborges  staff   128B Aug 24 10:54 com.apple.AMPDeviceDiscoveryAgent
drwx------@ 4 alexandreborges  staff   128B Jun  5  2023 com.apple.AVConference.Diagnostic
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.Accessibility-Settings.extension
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.ActionKit.BundledIntentHandler
drwx------@ 4 alexandreborges  staff   128B Jul 29 02:31 com.apple.AddressBook
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.AirDrop-Handoff-Settings.extension
drwx------@ 4 alexandreborges  staff   128B Aug 24 10:54 com.apple.AirPlayUIAgent
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.Animoji.StickersApp.MessagesExtension
drwx------@ 4 alexandreborges  staff   128B Jun  5  2023 com.apple.AppSSOKerberos.KerberosExtension
drwx------@ 4 alexandreborges  staff   128B Aug 25 00:39 com.apple.AppStore
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.AppStoreDaemon.ASDAskPermissionExtension
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.Appearance-Settings.extension
drwx------@ 4 alexandreborges  staff   128B Aug 24 10:35 com.apple.AppleAccountUI.AAUIFollowUpExtension
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.AppleMediaServices.FollowUpExtension
drwx------@ 4 alexandreborges  staff   128B Jun  1  2024 com.apple.AppleMediaServicesUI.ComposeReviewExtension
drwx------@ 4 alexandreborges  staff   128B Jun 10  2023 com.apple.AppleMediaServicesUI.EngagementViewExtension
alexandreborges@Mac Containers %
alexandreborges@Mac Containers % ls -lha com.apple.Safari/
total 112
drwx------@   4 alexandreborges  staff   128B Jun 27  2024 .
drwx------  464 alexandreborges  staff    15K Jan  3 21:55 ..
-rw-r--r--    1 alexandreborges  staff    55K Jun 27  2024 .com.apple.containermanagerd.metadata.plist
drwx------   13 alexandreborges  staff   416B Jun  5  2023 Data
alexandreborges@Mac Containers %
alexandreborges@Mac Containers % ls -lha com.apple.Safari/Data
total 0
drwx------  13 alexandreborges  staff   416B Jun  5  2023 .
drwx------@  4 alexandreborges  staff   128B Jun 27  2024 ..
lrwxr-xr-x   1 alexandreborges  staff    31B Jun  5  2023 .CFUserTextEncoding -> ../../../../.CFUserTextEncoding
drwxr-xr-x@  8 alexandreborges  staff   256B May  1  2024 CloudKit
lrwxr-xr-x   1 alexandreborges  staff    19B Jun  5  2023 Desktop -> ../../../../Desktop
drwx------   2 alexandreborges  staff    64B Jun  5  2023 Documents
lrwxr-xr-x   1 alexandreborges  staff    21B Jun  5  2023 Downloads -> ../../../../Downloads
drwx------  35 alexandreborges  staff   1.1K Jun  5  2023 Library
lrwxr-xr-x   1 alexandreborges  staff    18B Jun  5  2023 Movies -> ../../../../Movies
lrwxr-xr-x   1 alexandreborges  staff    17B Jun  5  2023 Music -> ../../../../Music
lrwxr-xr-x   1 alexandreborges  staff    20B Jun  5  2023 Pictures -> ../../../../Pictures
drwx------   2 alexandreborges  staff    64B Jun  5  2023 SystemData
drwx------   4 alexandreborges  staff   128B May  1  2024 tmp
alexandreborges@Mac Containers %
```

[Figure 95]: Sandbox containers

- As we can see in the figure above, there are a series of **symbolic links** and also a **plist** containing a set of properties that determine distinguished restrictions to operate withing the given container.

- An application has sandbox enabled and imposed through **com.apple.security.app-sandbox** entitlement, which makes part of the application's signature. Thus, such entitlement causes an automatic sandbox environment creation

- The first lines of the mentioned **plist**, which is binary, and respective lines of the **entitlement list**, including "sandbox" pattern, can be visualized by running:

```
alexandreborges@Mac Containers % cat com.apple.Safari/.com.apple.containermanagerd.me
tadata.plist| plutil -convert xml1 -o - - | head -20
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/Prope
rtyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>MCMMetadataActiveDPClass</key>
        <integer>-1</integer>
        <key>MCMMetadataContentClass</key>
        <integer>2</integer>
        <key>MCMMetadataIdentifier</key>
        <string>com.apple.Safari</string>
        <key>MCMMetadataInfo</key>
        <dict>
                <key>Identity</key>
                <array>
                        <data>
                        +t4MAAAAACwAAAABAAAABgAAAAIAAAAQY29tLmFwcGxlLlNhZmFy
                        aQAAAAM=
                        </data>
                </array>
                <key>LSDataContainerPersonalityInfo</key>
alexandreborges@Mac Containers %
alexandreborges@Mac Containers % codesign -d --entitlements - /Applications/Safari.ap
p/ | grep sandbox
Executable=/System/Volumes/Preboot/Cryptexes/App/System/Applications/Safari.app/Conte
nts/MacOS/Safari
        [Key] com.apple.private.RemoteServiceDiscovery.allow-sandbox
        [Key] com.apple.private.app-sandbox.unquarantine
        [Key] com.apple.private.safari.can-use-sandbox-broker
        [Key] com.apple.security.app-sandbox
                        [String] (allow file-issue-extension (require-all (extension-
class "com.apple.app-sandbox.read-write") (home-subpath "/Library/Cookies")))
                        [String] (with-filter (extension "com.apple.safari.develop-me
nu") (allow network-outbound (literal "/private/var/run/usbmuxd")) (allow file-issue-
extension (extension-class "com.apple.app-sandbox.read")) (allow network-outbound (re
gex #"com\.apple\.webinspectord_sim\.socket$")) (allow mach-lookup (global-name-regex
 #"^com\.apple\.CoreSimulator\.LaunchHost.*") (global-name #"com.apple.CoreSimulator.
simdiskimaged")))
alexandreborges@Mac Containers %
```

[Figure 96]: Sandboxed application plist (truncated)

- **Sandboxing**, which is a not an exclusive idea from macOS and iOS, acts as a second line of defense in case other protecting features (AMFI, for example) are not enough to deter threats. Thus, applications with **com.apple.security.app-sandbox** entitlement (as shown above in Safari case) are application sandbox-enabled, and only can access allowed system resources and data.

- Sandboxes have strict rules on resources that can be or are not accessed, but they can be adjusted through a **Sandbox profile** (.sb extension), and profiles from sandboxed applications can be mainly found in **/System/Library/Sandbox/Profiles** and **/usr/share/sandbox** directories:

```
alexandreborges@Mac ~ % ls -lh /System/Library/Sandbox/Profiles | head -10
total 3416
-rw-r--r--  1 root  wheel   4.1K Aug  4 07:31 ASPCarryLog.sb
-rw-r--r--  1 root  wheel   2.0K Aug  4 07:31 NANDTaskScheduler.sb
-rw-r--r--  1 root  wheel   2.0K Aug  4 07:31 NFRestoreService.sb
-rw-r--r--  1 root  wheel   2.9K Aug  4 07:31 WeatherKitAuthService.sb
-rw-r--r--  1 root  wheel   1.3K Aug  4 07:31 WiFiVelocityAgent.sb
-rw-r--r--  1 root  wheel    27K Aug  4 07:31 airlock.sb
-rw-r--r--  1 root  wheel    42K Aug  4 07:31 application.sb
-rw-r--r--  1 root  wheel    30K Aug  4 07:31 appsandbox-common.sb
-rw-r--r--  1 root  wheel   6.7K Aug  4 07:31 apsd.sb
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % ls -lh /usr/share/sandbox | head -10
total 536
-rw-r--r--  2 root  wheel   7.3K Aug  4 07:31 airportd.sb
-rw-r--r--  1 root  wheel   162B Aug  4 07:31 awdd.sb
-rw-r--r--  1 root  wheel    12K Aug  4 07:31 bluetoothd.sb
-rw-r--r--  1 root  wheel   9.7K Aug  4 07:31 com.apple.CommCenter.sb
-rw-r--r--  1 root  wheel   416B Aug  4 07:31 com.apple.FontValidator.sb
-rw-r--r--  1 root  wheel   2.4K Aug  4 07:31 com.apple.USBAgent.sb
-rw-r--r--  1 root  wheel   4.7K Aug  4 07:31 com.apple.atsd.internal.sb
-rw-r--r--  1 root  wheel   3.7K Aug  4 07:31 com.apple.atsd.support.sb
-rw-r--r--  1 root  wheel    18K Aug  4 07:31 com.apple.bird.sb
alexandreborges@Mac ~ %
```

[Figure 97]: Sandbox Profiles (truncated)

- A simple script can be useful to check for sandboxed processes:

```
alexandreborges@Mac ~ % cat check_sandbox.sh
#!/bin/bash

# Check if a PID is provided
if [ -z "$1" ]; then
 echo "Usage: $0 <pid>"
 exit 1
fi

# Get the path to the process from the PID
process_path=$(ps -p $1 -o comm=)

# Check if the process path is found
if [ -z "$process_path" ]; then
  echo "Process with PID $1 not found."
  exit 1
fi

# Extract the entitlements and check for sandbox
entitlements=$(codesign -d --entitlements - --xml $process_path 2>/dev/null)
process_name=$(basename $process_path 2>/dev/null)
if echo "$entitlements" | grep -q "com.apple.security.app-sandbox"; then
  echo "The process $process_name (PID $1) is sandboxed."
else
  echo "The process $process_name (PID $1) is not sandboxed."
fi
```

[Figure 98]: Bash script to check for sandbox processes

- Evaluating the script against different processes:

```
alexandreborges@Mac ~ % ps aux | grep logd
root               947   0.0  0.1 33691404   2140   ??  Ss   29Oct24   0:00.95 /usr/
libexec/logd_reporter
root               417   0.0  0.0 33658324   1044   ??  Ss   29Oct24   0:00.64 /usr/
libexec/logd_helper
root               123   0.0  0.0 33659252    832   ??  Ss   29Oct24   0:07.07 /usr/
sbin/syslogd
root                86   0.0  1.0 33825788  43140   ??  Ss   29Oct24   5:32.49 /usr/
libexec/logd
alexandreborges  65015   0.0  0.0 34121188    540 s001  S+    1:31AM   0:00.00 grep
logd
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % ps aux | grep -i message
alexandreborges    695   0.0  0.1 33700524   6260   ??  Ss   29Oct24   0:00.32 /Syst
em/Applications/Messages.app/Contents/Extensions/MessagesActionExtension.appex/Conte
nts/MacOS/MessagesActionExtension
alexandreborges  65017   0.0  0.0 34125288    468 s001  R+    1:31AM   0:00.00 grep
-i message
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % ./check_sandbox.sh 86
The process logd (PID 86) is not sandboxed.
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % ./check_sandbox.sh 695
The process MessagesActionExtension (PID 695) is sandboxed.
alexandreborges@Mac ~ %
```

[Figure 99]: Testing the script that checks whether a process is or is not sandboxed.

- If you want, you can check for all processes to discover those ones that are using sandbox:

```
alexandreborges@Mac ~ % ps -A -o pid | tail -n +2 | while read -r pid; do ./che
ck_sandbox.sh $pid | grep "is sandboxed"; done | head -15
The process com.apple.geod (PID 241) is sandboxed.
The process com.apple.AmbientDisplayAgent (PID 376) is sandboxed.
The process com.apple.hiservices-xpcservice (PID 397) is sandboxed.
The process com.apple.hiservices-xpcservice (PID 442) is sandboxed.
The process WallpaperAgent (PID 443) is sandboxed.
The process routined (PID 455) is sandboxed.
The process WallpaperImageExtension (PID 457) is sandboxed.
The process QuickLookUIService (PID 462) is sandboxed.
The process IOUIAgent (PID 469) is sandboxed.
The process AMPDeviceDiscoveryAgent (PID 477) is sandboxed.
The process NotificationCenter (PID 493) is sandboxed.
The process AirPlayUIAgent (PID 508) is sandboxed.
The process TextInputSwitcher (PID 521) is sandboxed.
The process com.apple.CloudDocs.iCloudDriveFileProvider (PID 531) is sandboxed.
The process CalendarWidgetExtension (PID 543) is sandboxed.
alexandreborges@Mac ~ %
```

[Figure 100]: Checking multiple processes for sandbox

- There are many components that make part of the Sandboxing mechanism in userland such as **/usr/libexec/sandboxd** and **/usr/libexec/containermanagerd** (daemons) and some libraries such as **/usr/lib/libsystem_sandbox.dylib** and **/usr/lib/libsandbox.1.dylib** (there are other ones), as well as in kernel-side such as **Sandbox.kext** and **AppleMatch.ext**, for example.

- The decision whether an application should or not be secured by a Sandbox is from **secinitd** daemon, which uses XPC as communication mechanism (different from **sandboxd**, which communicates via Mach). In iOS applications are usually isolated within a container and its own location is an indicator for the system whether it must or not be installed inside a sandbox (**/var/container** and **/var/mobile/Container**). Anyway, the key role of a sandbox is to intercept requests for resources, interpret them and decide if such a request is or is not allowed. **Profiles** are inserted in this equation to make decisions more flexible and adaptable to distinguish circumstances.

- Only to show as **AMFI, Sandbox, quarantine** and **policies** are integrated to **MACF mechanism**, readers notice that all of them register themselves as a **MACF policy**, as shown below:

```
alexandreborges@alexandremacos all % disarm -S com.apple.driver.AppleMobileFileI
ntegrity | grep _mac_policy
                 U _mac_policy_register
fffffe0009d5f87c T _amfi_register_mac_policy
alexandreborges@alexandremacos all % disarm -S com.apple.security.sandbox| grep
_mac_policy
                 U _mac_policy_register
                 U _amfi_register_mac_policy
alexandreborges@alexandremacos all % disarm -S com.apple.security.quarantine| gr
ep _mac_policy
                 U _mac_policy_register
alexandreborges@alexandremacos all % disarm -S com.apple.AppleSystemPolicy| grep
 _mac_policy
                 U _mac_policy_register
alexandreborges@alexandremacos all %
```

[Figure 101]: Different components registering themselves as MACF policy.

- **SIP (System Integrity Protection)** is the last piece of the main puzzle (of course, there are other frameworks that work in a different context). **SIP** is a necessary protection (at least, in my opinion), which limits even the root user's actions in changing important directories and files, prevents debugging sections of protected system) process, prevents the loading of kernel extension and other activities. Of course, it is not good for us vulnerability researchers, but it is necessary for the integrity of the ecosystem.

- SIP is initiated from the boot when the **launchd** invokes **/usr/libexec/rootless-init**, which uses **/System/Library/Sandbox/rootless.conf** as configuration file that determines files and directories to be protected.

- The operating system marks protected objects through an attribute named com.apple.rootless, which can be visualized using **ls -l@** command. or **restricted flag** that can be checked by running **ls -ld0@.**

- SIP can be disabled by booting the macOS in Recovery Mode, executing a Terminal and then running **csrutil disable** command. To check the current configuration, execute **csrutil status**. Yes, there is a "hidden" option using **--without** option, but it is not necessary to discuss it now.

```
alexandreborges@Mac ~ % cat /System/Library/Sandbox/rootless.conf
                                /Applications/Safari.app
                                /Library/Apple
CoreAnalytics                   /Library/CoreAnalytics
NetFSPlugins                    /Library/Filesystems/NetFSPlugins/Staged
NetFSPlugins                    /Library/Filesystems/NetFSPlugins/Valid
                                /Library/Frameworks/iTunesLibrary.framework
KernelExtensionManagement       /Library/GPUBundles
KernelExtensionManagement       /Library/KernelCollections
MessageTracer                   /Library/MessageTracer
AudioSettings                   /Library/Preferences/Audio/Data
                                /Library/Preferences/SystemConfiguration/com.apple.Boot
.plist
KernelExtensionManagement       /Library/StagedDriverExtensions
KernelExtensionManagement       /Library/StagedExtensions
SoftwareUpdate                  /Library/Updates
                                /System
*                               /System/Applications/Safari.app/Contents/MacOS/SafariFo
rWebKitDevelopment
MobileStorageMounter            /System/Developer
MobileAsset                     /System/Library/Assets
MobileAsset                     /System/Library/AssetsV2
*                               /System/Library/Caches
KernelExtensionManagement       /System/Library/Caches/com.apple.kext.caches
*                               /System/Library/Extensions
                                /System/Library/Extensions/*
UpdateSettings                  /System/Library/LaunchDaemons/com.apple.UpdateSettings.
plist
MobileAsset                     /System/Library/PreinstalledAssets
MobileAsset                     /System/Library/PreinstalledAssetsV2
*                               /System/Library/Speech
# template locations
*                               /System/Library/Templates/Data
                                /System/Library/Templates/Data/Applications/Safari.app
                                /System/Library/Templates/Data/Library/Apple
CoreAnalytics                   /System/Library/Templates/Data/Library/CoreAnalytics
NetFSPlugins                    /System/Library/Templates/Data/Library/Filesystems/NetF
SPlugins/Staged
NetFSPlugins                    /System/Library/Templates/Data/Library/Filesystems/NetF
SPlugins/Valid
                                /System/Library/Templates/Data/Library/Frameworks/iTune
sLibrary.framework
KernelExtensionManagement       /System/Library/Templates/Data/Library/GPUBundles
KernelExtensionManagement       /System/Library/Templates/Data/Library/KernelCollection
s
MessageTracer                   /System/Library/Templates/Data/Library/MessageTracer
```

[Figure 102]: rootless.conf file (truncated) -- protected files and directories

```
alexandreborges@Mac ~ % ls -l@ /Applications/Safari.app
lrwxr-xr-x@ 1 root  wheel  54 Aug  4 07:31 /Applications/Safari.app -> ..
/System/Cryptexes/App/System/Applications/Safari.app
        com.apple.rootless      0
alexandreborges@Mac ~ %
alexandreborges@Mac ~ % ls -lO@ /Applications/Safari.app
lrwxr-xr-x@ 1 root  wheel  restricted,hidden 54 Aug  4 07:31 /Application
s/Safari.app -> ../System/Cryptexes/App/System/Applications/Safari.app
        com.apple.rootless      0
alexandreborges@Mac ~ %
```

[Figure 103]: Checking marks of a restricted (protected) file

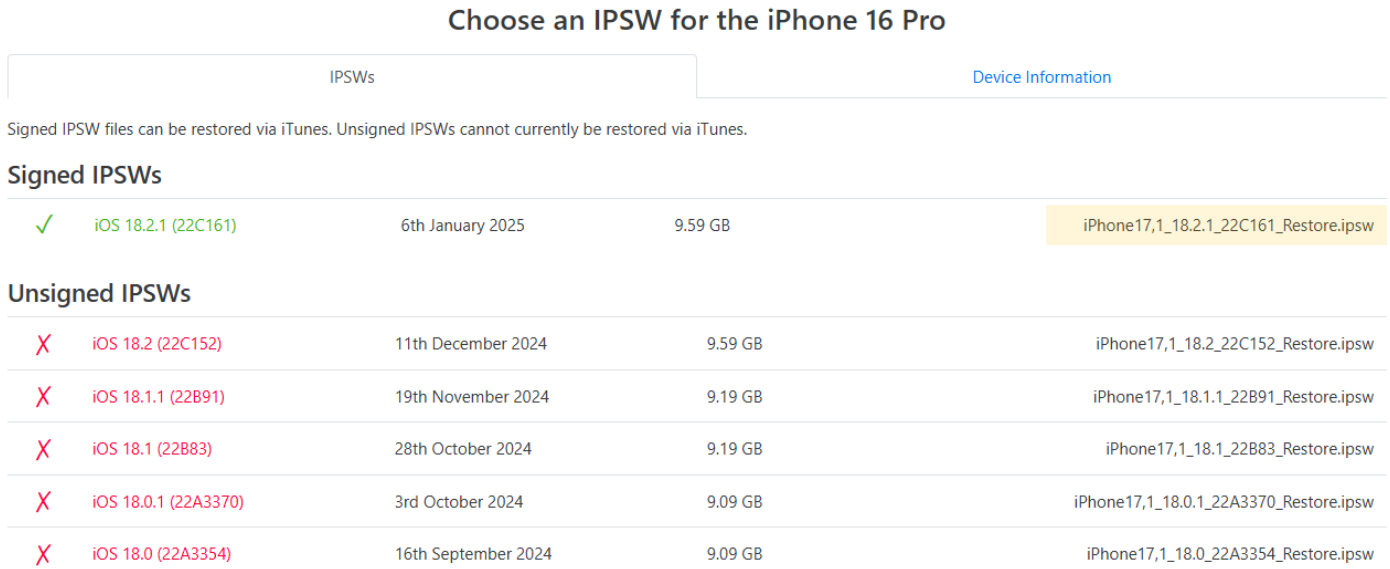
- SIP is an effective protection, but only for those protected objects and nothing more. Therefore, SIP will not protect against any well-known vulnerability class or any other vulnerability. Additionally, there are entitlements such as **com.apple.rootless.install.heritable** and **com.apple.rootless.install** that if granted to a user, so there is a possibility to circumvent SIP mechanism.

I think that shortened facts on macOS listed so far is enough to review the main points of its architecture. Of course, I could have mentioned details about privacy, keychains, and other topics, but they would be out of context for now and, eventually, I will return to these subjects in the next articles.

09. Additional words on the kernelcache

In a few words, kernelcache is a compressed and signed file that contains the macOS or iOS kernel, and its respective extensions. As mentioned, in past versions of macOS/iOS the kernelcache was encrypted, but it is no longer the case. Previously in this text I have already shown how to download, uncompress and even open the kernelcache on IDA Pro, but even so I will repeat a few of these steps for completeness, but this time adopting a manual approach.

You can proceed using multiples approaches here. For example, you can access the https://ipsw.me/product/iPhone website, choose an iPhone model and then get the exact name of the IPSW file.



[Figure 104]: Details about a given IPSW file from IPSW website

As usual, Apple keeps the last firmware updated, and all the old ones automatically become unsigned, which prevents to apply old firmware applying through the official form. Actually, I have mentioned this because it is one of best sources of Apple device's firmware and, mainly, because it provides an entire set of APIs to interact with the website that is available on https://ipswdownloads.docs.apiary.io/# for list devices, download firmware and much more. It is worth reading its documentation.

Readers can write their own script to retrieve the needed information and, in this case, here is an example script provides you with both a list of devices, firmware and also downloads the given firmware:

```
import requests
import sys
import os
import time
import signal

def get_device_info(identifier):
    url = f"https://api.ipsw.me/v4/device/{identifier}?type=ipsw"
    response = requests.get(url)

    if response.status_code == 200:
        return response.json()
    else:
        print(f"Error: Unable to fetch data for identifier {identifier}. Status code: {response.st
atus_code}")
        return None

def get_all_devices():
    url = "https://api.ipsw.me/v4/devices"
    response = requests.get(url)

    if response.status_code == 200:
        return response.json()
    else:
        print(f"Error: Unable to fetch all devices. Status code: {response.status_code}")
        return None

def download_firmware(identifier, buildid):
    url = f"https://api.ipsw.me/v4/ipsw/download/{identifier}/{buildid}"
    filename = f"{identifier}_{buildid}.ipsw"
    headers = {}

    if os.path.exists(filename):
        existing_file_size = os.path.getsize(filename)
        headers = {'Range': f'bytes={existing_file_size}-'}
    else:
        existing_file_size = 0

    response = requests.get(url, headers=headers, stream=True)
    total_size = int(response.headers.get('content-length', 0)) + existing_file_size
    block_size = 8192  # 8 KB
    progress_bar_size = 50  # Length of the progress bar

    def signal_handler(sig, frame):
        print("\nDownload cancelled")
        sys.exit(0)

    signal.signal(signal.SIGINT, signal_handler)

    if response.status_code in [200, 206]:  # 206 indicates a partial content response
        mode = 'ab' if existing_file_size else 'wb'
        with open(filename, mode) as file:
            start_time = time.time()
            for data in response.iter_content(block_size):
                file.write(data)
                progress = (file.tell() / total_size)
                progress_bar = '=' * int(progress * progress_bar_size)
                percent_complete = int(progress * 100)
                elapsed_time = time.time() - start_time
                estimated_total_time = elapsed_time / progress
```

```
                estimated_remaining_time = (estimated_total_time - elapsed_time) / 60  # Convert seconds to minutes
                print(f"\r[{'='*int(progress * progress_bar_size):<{progress_bar_size}}] {percent_complete}% ({estimated_remaining_time:.2f} minutes remaining)", end='')

        print(f"\nFirmware downloaded successfully as {filename}")
    else:
        print(f"Error: Unable to download firmware for identifier {identifier} and build ID {buildid}. Status code: {response.status_code}")

def display_device_info(device_info):
    if device_info:
        print(f"{'Device Name':<20}: {device_info.get('name')}")
        print(f"{'Identifier':<20}: {device_info.get('identifier')}")
        print("\nFirmwares:")
        print(f"{'Version':<10} {'Build ID':<10} {'Signed':<10}")
        print(f"{'-'*10} {'-'*10} {'-'*10}")
        for firmware in device_info.get('firmwares', []):
            print(f"{firmware.get('version'):<10} {firmware.get('buildid'):<10} {str(firmware.get('signed')):<10}")
    else:
        print("No information available.")

def display_all_devices(devices):
    if devices:
        devices = devices[::-1]  # Reverse the order of devices
        print(f"{'Device Name':<40} {'Identifier':<20}")
        print(f"{'-'*40} {'-'*20}")
        for device in devices:
            print(f"{device['name']:<40} {device['identifier']:<20}")
    else:
        print("No information available.")

def show_help():
    print("Usage:")
    print("  python script.py                          - Show this help message")
    print("  python script.py --listdevices            - List all devices")
    print("  python script.py --firmware <identifier> - Show firmware details for a specific device identifier")
    print("  python script.py --download <identifier> <buildid> - Download firmware for a specific device identifier and build ID")
    print("  python script.py -h                       - Show this help message")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        show_help()
    elif sys.argv[1] in ['-h', '--help']:
        show_help()
    elif sys.argv[1] == '--listdevices':
        devices = get_all_devices()
        display_all_devices(devices)
    elif sys.argv[1] == '--firmware' and len(sys.argv) == 3:
        identifier = sys.argv[2]
        device_info = get_device_info(identifier)
        display_device_info(device_info)
    elif sys.argv[1] == '--download' and len(sys.argv) == 4:
        identifier = sys.argv[2]
        buildid = sys.argv[3]
        download_firmware(identifier, buildid)
    else:
        show_help()

```

[Figure 105]: sample script that implements APIs from IPSW Downloads website

Assessing the script we have the following output:

```
alexandreborges@alexandremacos kernelcache % python3 script.py
Usage:
  python script.py                          - Show this help message
  python script.py --listdevices            - List all devices
  python script.py --firmware <identifier> - Show firmware details for a specific device identifier
  python script.py --download <identifier> <buildid> - Download firmware for a specific device identif
ier and build ID
  python script.py -h                       - Show this help message
alexandreborges@alexandremacos kernelcache %
alexandreborges@alexandremacos kernelcache % python3 script.py --listdevices | grep iPhone | head -10
iPhone 16 Pro                             iPhone17,1
iPhone 16                                 iPhone17,3
iPhone 16 Pro Max                         iPhone17,2
iPhone 16 Plus                            iPhone17,4
iPhone 15 Pro Max                         iPhone16,2
iPhone 15 Pro                             iPhone16,1
iPhone 15 Plus                            iPhone15,5
iPhone 15                                 iPhone15,4
iPhone 14 Pro Max                         iPhone15,3
iPhone 14 Pro                             iPhone15,2
alexandreborges@alexandremacos kernelcache % python3 script.py --firmware iPhone17,3
Device Name          : iPhone 16
Identifier           : iPhone17,3

Firmwares:
Version    Build ID   Signed
---------- ---------- ----------
18.2.1     22C161     True
18.2       22C152     False
18.1.1     22B91      False
18.1       22B83      False
18.0.1     22A3370    False
18.0       22A3354    False
alexandreborges@alexandremacos kernelcache % python3 script.py --download iPhone17,3 22C161
[===========                                        ] 22% (4.02 minutes remaining)
```

[Figure 106]: output from the sample script

We can create a new subdirectory (unpacking/) and move the downloaded IPSW file into it. Change to the subdirectory, rename the file to add the **.zip suffix** and uncompress it:

```
alexandreborges@alexandremacos unpacked % ls
iPhone17,3_22C161.ipsw
alexandreborges@alexandremacos unpacked % mv iPhone17,3_22C161.ipsw iPhone17,3_22C161.ipsw.zip
alexandreborges@alexandremacos unpacked %
alexandreborges@alexandremacos unpacked % unzip iPhone17,3_22C161.ipsw.zip
Archive:  iPhone17,3_22C161.ipsw.zip
 extracting: 092-09077-213.dmg.aea
  inflating: 092-09241-225.dmg
  inflating: 092-09301-225.dmg
 extracting: 092-09474-225.dmg.aea
 extracting: 092-09894-205.dmg.aea
  inflating: 092-09979-225.dmg
 extracting: 092-10153-224.dmg.aea
  inflating: BuildManifest.plist
   creating: Firmware/
  inflating: Firmware/092-09077-213.dmg.aea.root_hash
  inflating: Firmware/092-09077-213.dmg.aea.trustcache
  inflating: Firmware/092-09241-225.dmg.trustcache
```

[Figure 107]: uncompressing the IPSW file

The result of the uncompressing task follows below:

```
alexandreborges@alexandremacos unpacked % ls -lh
total 36716904
-rw-r--r--   1 alexandreborges  staff   1.5G Jan  9  2007 092-09077-213.dmg.aea
-rw-r--r--   1 alexandreborges  staff   146M Jan  9  2007 092-09241-225.dmg
-rw-r--r--   1 alexandreborges  staff   144M Jan  9  2007 092-09301-225.dmg
-rw-r--r--   1 alexandreborges  staff   160M Jan  9  2007 092-09474-225.dmg.aea
-rw-r--r--   1 alexandreborges  staff   6.4G Jan  9  2007 092-09894-205.dmg.aea
-rw-r--r--   1 alexandreborges  staff    14M Jan  9  2007 092-09979-225.dmg
-rw-r--r--   1 alexandreborges  staff   212M Jan  9  2007 092-10153-224.dmg.aea
-r--r--r--   1 alexandreborges  staff   847K Jan  9  2007 BuildManifest.plist
drwxr-xr-x  53 alexandreborges  staff   1.7K Jan  9  2007 Firmware
-r--r--r--   1 alexandreborges  staff   1.1K Jan  9  2007 Restore.plist
-r--r--r--   1 alexandreborges  staff   360B Jan  9  2007 RestoreVersion.plist
-r--r--r--   1 alexandreborges  staff   492B Jan  9  2007 SystemVersion.plist
-rw-r--r--   1 alexandreborges  staff   8.8G Jan 22 00:07 iPhone17,3_22C161.ipsw.zip
-rwxr--r--   1 alexandreborges  staff    19M Jan  9  2007 kernelcache.release.iphone17
-rwxr--r--   1 alexandreborges  staff    19M Jan  9  2007 kernelcache.research.iphone17
alexandreborges@alexandremacos unpacked %
alexandreborges@alexandremacos unpacked % file kernelcache.release.iphone17
kernelcache.release.iphone17: data
alexandreborges@alexandremacos unpacked % ipsw kernel dec kernelcache.release.iphone17
   • Decompressing kernelcache
      • Created kernelcache.release.iphone17.decompressed
alexandreborges@alexandremacos unpacked %
alexandreborges@alexandremacos unpacked % file kernelcache.release.iphone17.decompressed
kernelcache.release.iphone17.decompressed: Mach-O 64-bit arm64e
alexandreborges@alexandremacos unpacked %
alexandreborges@alexandremacos unpacked % nm kernelcache.release.iphone17.decompressed
kernelcache.release.iphone17.decompressed: no symbols
alexandreborges@alexandremacos unpacked %
```

[Figure 108]: List of extracted files and directory from an IPSW file

We got our **kernelcache** at the end of the list above. However, as it was **compressed**, as we have learned previously (pay attention the file type is **data**), we have **decompress** it using **ipsw kernel dec** command (we could use **img4tool** to do it) and finally we got a **Mach-O 64-bit arm64e** file that, as expected, **does not have any symbol** according to **nm** command. By the way, the kernelcache is not digitally signed. We can list the kexts (kernel extensions) within the kernelcache by executing the following command:

```
alexandreborges@alexandremacos unpacked % ipsw kernel kexts kernelcache.release.iphone17.
decompressed | head -10
   • Kexts                    count=310
0x7fffffffffffff: com.apple.driver.AppleBSDKextStarterTMPFS (1)
0x7fffffffffffff: com.apple.driver.AppleBSDKextStarterVPN (3)
0x7fffffffffffff: com.apple.driver.AppleHIDKeyboardEmbedded (1.2.0a3)
0x7fffffffffffff: com.apple.driver.AppleStorageDrivers (556)
0x7fffffffffffff: com.apple.driver.AppleThunderboltEDMService (5.0.3)
0x7fffffffffffff: com.apple.driver.AppleTopCaseDriverV2 (8100.22)
0x7fffffffffffff: com.apple.driver.SCSIDeviceSpecifics (556)
0x7fffffffffffff: com.apple.driver.USBStorageDeviceSpecifics (556)
0x7fffffffffffff: com.apple.driver.usb.AppleUSBHostPlatformProperties (1.2)
0x7fffffffffffff: com.apple.driver.usb.IOUSBHostHIDDeviceSafeBoot (1.2)
alexandreborges@alexandremacos unpacked %
```

[Figure 109]: List of kernel extensions (kexts) inside of a kernel cache

I have shown a script to check and download the firmware only to show you other alternatives, but readers could have done the same steps as I did previously in this article. Only to review steps again:

- **download the firmware: ipsw download ipsw --device iPhone17,3 --build 22C161**
- **extract and decompress the kernelcache: ipsw extract --kernel iPhone17,3_22C161.ipsw**
- **check the latest version: ipsw download ipsw --show-latest-version**

Returning to our task, if we want to extract one or all kernel extensions, execute:

- **extract all kernel extensions: ipsw kernel extract kernelcache.release.iphone17.decompressed --all --output kernel_extensions**
- **extract only one chosen kernel extension: ipsw kernel extract kernelcache.release.iphone17.decompressed IONetworkFamily**

To confirm the kernel version and also another way (and better) to check existing symbols run the following commands:

- **ipsw kernel version kernelcache.release.iphone17.decompressed --json | jq .**
- **ipsw macho info --fileset-entry com.apple.kernel kernelcache.release.iphone17.decompressed | grep LC_SYMTAB**

```
alexandreborges@alexandremacos unpacked % ipsw kernel version kernelcache.release.iphone17.
decompressed --json | jq .
{
  "kernel": {
    "darwin": "24.2.0",
    "date": "2024-11-14T22:55:26Z",
    "xnu": "11215.62.3~1",
    "type": "RELEASE",
    "arch": "ARM64",
    "cpu": "T8140"
  },
  "llvm": {}
alexandreborges@alexandremacos unpacked %
alexandreborges@alexandremacos unpacked % ipsw macho info --fileset-entry com.apple.kernel
kernelcache.release.iphone17.decompressed | grep LC_SYMTAB
020: LC_SYMTAB                  Symbol offset=0x03EDC000, Num Syms: 0, String offset=0x03F
3F838-0x03F3F839
alexandreborges@alexandremacos unpacked %
```

[Figure 110]: Checking general information and symbols within a kernel cache

From this point onward, we have a couple of alternatives to bring symbols to the kernelcache:

- Use **IDA Pro + Lumina**, which will do all the job in an excellent way. I have already shown something about it previously, but I will repeat steps again because, in the end, it is my preferred approach.

- Use a project like **Symbolicator tool** (https://github.com/blacktop/symbolicator), which does a great job of symbolicate the **kernelcache**, and then there is a plugin to integrate all retrieved symbols to the code.

To cover both approaches, I start using **Symbolicator tool** and it is necessary to clone the project:

- **git clone https://github.com/blacktop/symbolicator.git**

To symbolicate the kernelcache, execute:

- **ipsw kernel symbolicate kernelcache.release.iphone17.decompressed --json --signatures ../../github/symbolicator/kernel**

```
alexandreborges@alexandremacos unpacked % ipsw kernel symbolicate kernelcache.release.iphone17
.decompressed --json --signatures ../../github/symbolicator/kernel
   • Parsing Signatures
   • Symbolicating...          kernelcache=kernelcache.release.iphone17.decompressed
   • Found                     bsd_syscall_table=0xfffffff007b44e20
   • Found                     mach_trap_table=0xfffffff007b1c028
   • Found                     mig_kern_subsystem table=0xfffffff00abf4b08
   • Analyzing MachO...        name=com.apple.AUC
      • Symbolicated              address=0xfffffff00895c84c file=com.apple.AUC symbol=__ZN3AU
C5startEP9IOService
      • Symbolicated              address=0xfffffff00895ce18 file=com.apple.AUC symbol=__ZN3AU
C4freeEv
      • Symbolicated              address=0xfffffff00895d378 file=com.apple.AUC symbol=__ZN3AU
C30checkForHooverProtocolRequiredEv
         • Symbolicated (Caller)     address=0xfffffff00895d378 file=com.apple.AUC symbol=__ZN
3AUC26DPPluggedNotificationGatedEP9IOService
         • Symbolicated (Caller)     address=0xfffffff00895d378 file=com.apple.AUC symbol=__ZN
3AUC31call_DPPluggedNotificationGatedEP9IOService
      • Symbolicated              address=0xfffffff00895c654 file=com.apple.AUC symbol=__ZN3AU
C25AUCVideoInterfaceMatchingE12IOAVLocation
         • Symbolicated (Caller)     address=0xfffffff00895c654 file=com.apple.AUC symbol=__ZN
3AUC26createDisplayNotificationsEv
         • Symbolicated (Caller)     address=0xfffffff00895c654 file=com.apple.AUC symbol=__ZN
3AUC5startEP9IOService
      • Symbolicated              address=0xfffffff00895d794 file=com.apple.AUC symbol=__ZN3AU
C22InterfaceStatusAndTypeEP18IOAVVideoInterfacePjS2_
      • Symbolicated              address=0xfffffff00895c6cc file=com.apple.AUC symbol=__ZN3AU
      • Symbolicated              address=0xfffffff00860d1e0 file=com.apple.kernel symbol=net_f
ilter_event_enqueue_callback
         • Symbolicated (Caller)     address=0xfffffff00860d1e0 file=com.apple.kernel symbol=ne
t_filter_event_mark
      • Symbolicated              address=0xfffffff008630340 file=com.apple.kernel symbol=fsw_u
ninit
         • Symbolicated (Caller)     address=0xfffffff008630340 file=com.apple.kernel symbol=nx
_fsw_dom_terminate
      • Symbolicated              address=0xfffffff008786884 file=com.apple.kernel symbol=__ZN2
0IOExtensiblePaniclog14createWithUUIDEPhPKcj29ext_paniclog_create_options_tPPS_
   • 📈 Symbolication STATS    matched=22458 missed=4453 percent=83.4529% total=26911
   • Writing symbols as JSON to kernelcache.release.iphone17.decompressed.symbols.json
alexandreborges@alexandremacos unpacked %
```

[Figure 111]: Symbolicating the kernelcache (truncated)

The output above is a composition of its beginning and its ending and actually is much longer than it, and the result is saved into a JSON file. The symbols have been applied to 83.45% of the code.

Our next steps are the following:

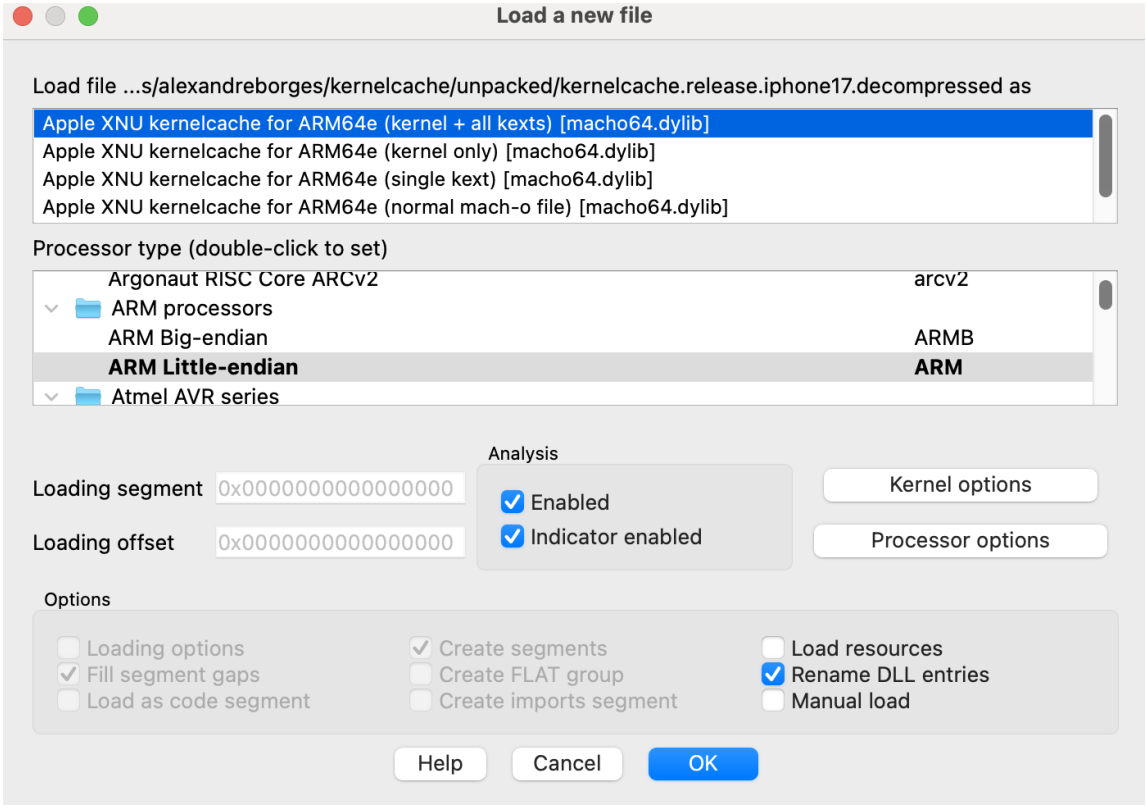
- Install the Symbolicate IDA Pro plugin (there are versions for Binary Ninja, Ghidra and Radare2)
- Open up the kernelcache on IDA Pro
- Apply the symbols (**ALT+F8**)

To install Symbolicate plugin execute the following steps:

```
alexandreborges@alexandremacos plugins % pwd
/Users/alexandreborges/github/symbolicator/plugins
alexandreborges@alexandremacos plugins %
alexandreborges@alexandremacos plugins % ls -lh
total 8
-rw-r--r--  1 alexandreborges  staff   170B Jan 23 00:45 README.md
drwxr-xr-x  8 alexandreborges  staff   256B Jan 23 00:45 binja
drwxr-xr-x  6 alexandreborges  staff   192B Jan 23 00:45 ghidra
drwxr-xr-x  7 alexandreborges  staff   224B Jan 23 00:45 ida
alexandreborges@alexandremacos plugins %
alexandreborges@alexandremacos plugins % mkdir -p ../../../.idapro/plugins
alexandreborges@alexandremacos plugins %
alexandreborges@alexandremacos plugins % ida/install.sh
  🚀 Installing /Users/alexandreborges/github/symbolicator/plugins/ida/symbolicate.py
 to /Users/alexandreborges/.idapro/plugins/
alexandreborges@alexandremacos plugins %
```

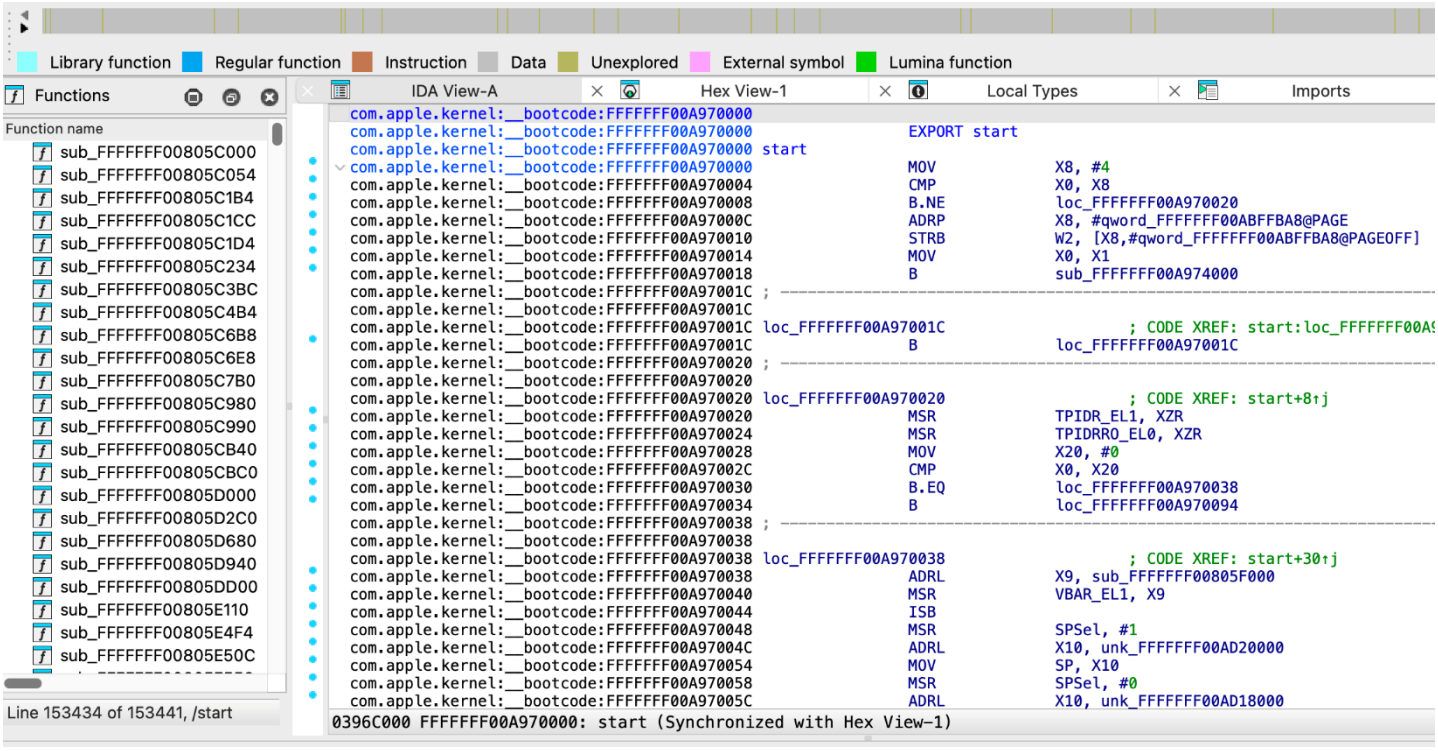
[Figure 112]: Installing symbolicator's IDA Pro plugin

Open the decompressed version of the kernelcache in IDA Pro, which would decompress it whether we had tried the compressed version, and you will see the following window:



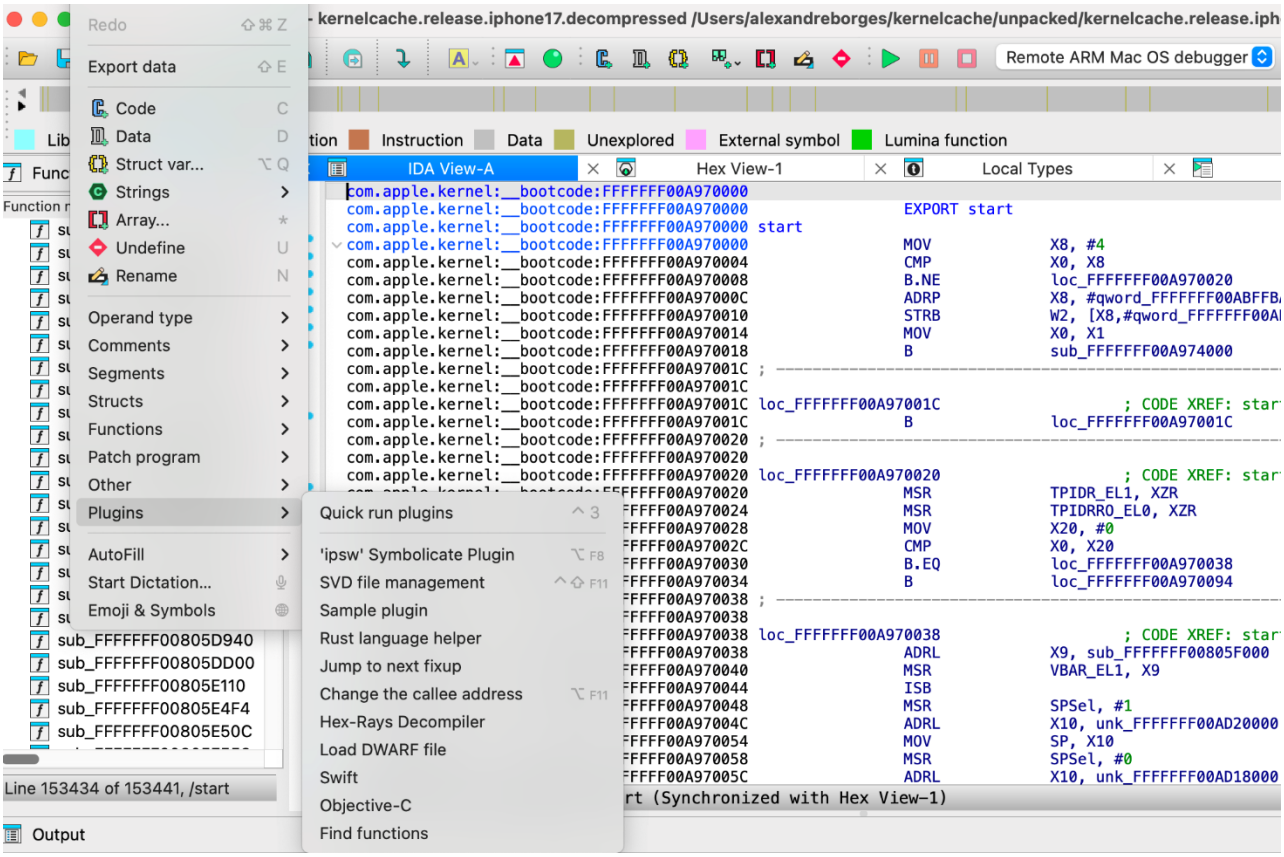
[Figure 113]: Open the decompressed kernelcache on IDA Pro

There are four distinct options here: load kernel + all kexts, load only the kernel, load only a kext or even interpret the kernelcache as a normal Mach-O file. In this case I will pick up the first option, which takes some time to finish the analysis, but the result is more complete. Once the first analysis occurs, we will the following:



[Figure 114]: Decompressed kernelcache on IDA Pro

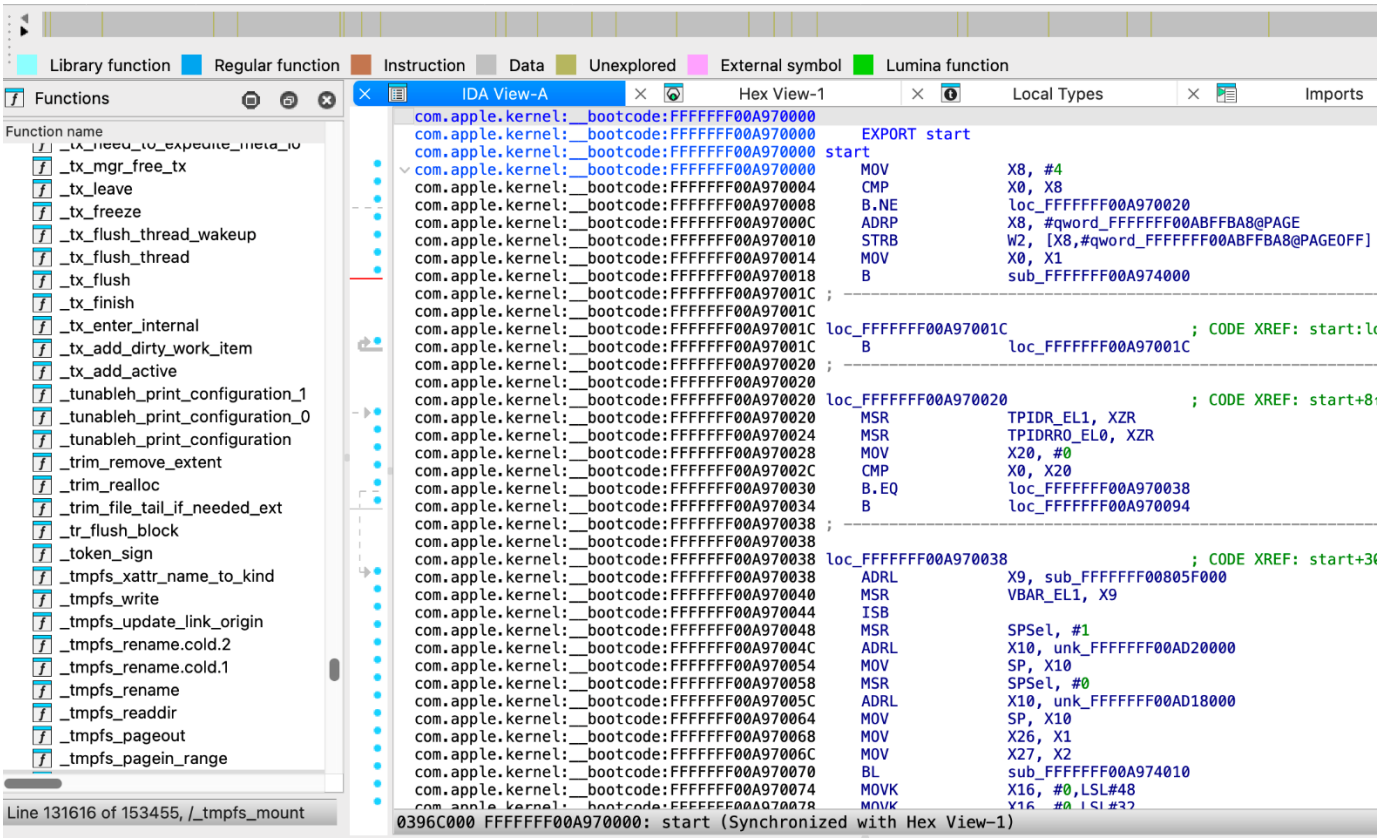
Now go to **Edit → Plugins → 'ipsw' Symbolicate Plugin**, as shown below:



[Figure 115]: Accessing the Symbolicate plugin

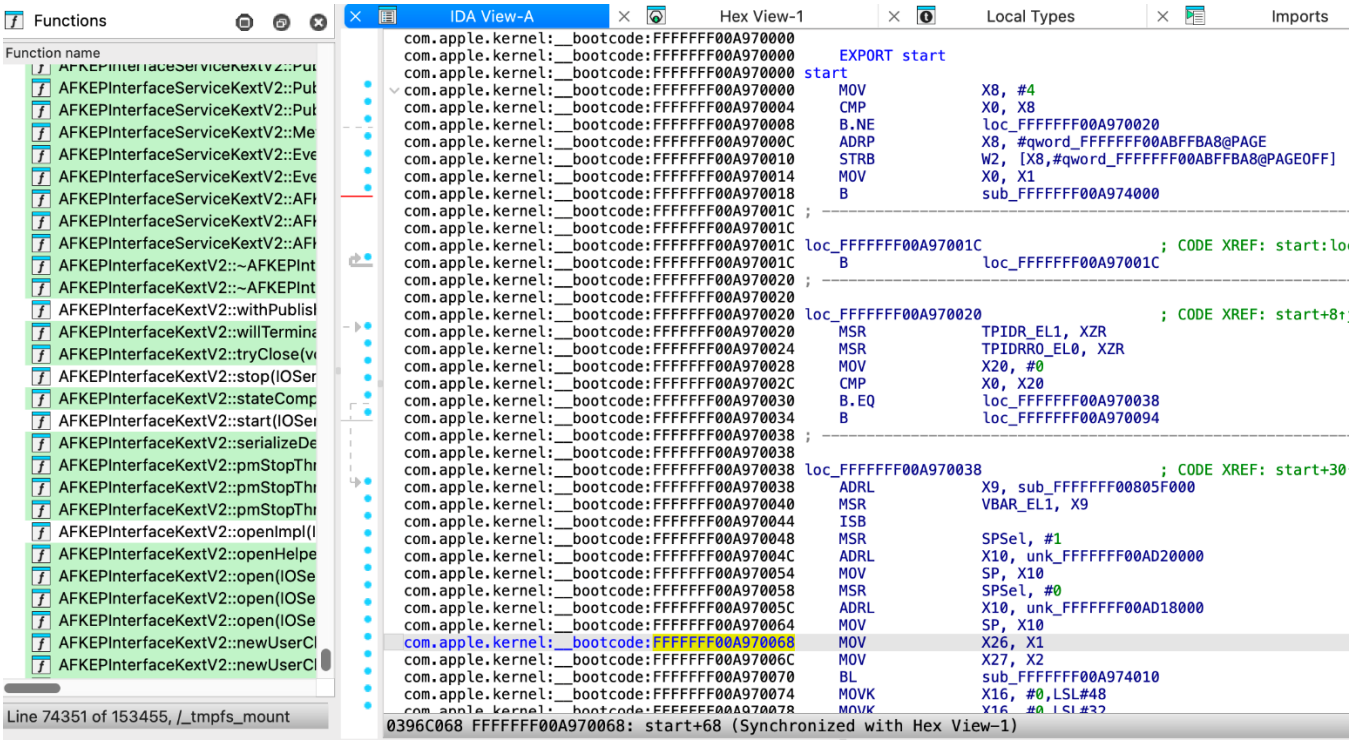
You will be prompted to choose the kernelcache.release.iphone17.decompressed.symbols.json file.

Once the file is picked up, all found symbols are applied, as shown below:



[Figure 116]: Symbols applied by IPSW Symbolicate's plugin

Preferably, if we apply symbols directly from Lumina (**Lumina → Pull all**), we have the final result:



[Figure 117]: Symbols applied by Lumina

Readers have realized that, at the first load of kernelcache in IDA, it takes a considerable amount of time to finish. One of alternatives is to do this process on the command line by executing:

- **TVHEADLESS=1 /Applications/IDA\ Pro\ 8.4/idabin/idat64 -Lida.log -B kernelcache.release.iphone17.decompressed**

There are two excellent projects that help the analysis of the kernelcache a lot, but for now they are updated up to iOS 17.0b1 and IDA 8.3. However, from time to time, these projects are updated, and the recommendation is to follow both:

- https://github.com/cellebrite-labs/ida_kernelcache
- https://github.com/cellebrite-labs/ida_kcpp

For now, we can stop our brief review about initial procedures to analyze the kernelcache.

10. Notes on iOS/macOS security and protections

I have avoided commenting a bit more about iOS (debugging is slightly different) in special (has similar pieced of code to macOS, but far from being identical), but there are reasons for following this line. It is not so simple to set up a lab because it depends on jailbroken mobile devices (using **untethered, semi-tethered** or **tethered** jailbreaking software) and clearly many readers might not have access to such hardware, which is the opposite of macOS, which readers can install it in a virtual machine for educational purposes. However, I will try not to make this distinction from this point onward (mainly in future articles).

From my point of view, getting a full exploitation of an iOS device (iPhone) is a real and hard challenge in the current days, many when executed through zero-click or one-click exploits. Eventually, bypassing a single protection like SIP could not be difficult (everything depends on the context), but full exploitation is not composed by a single task, but of a set of exploits to circumvent different protections, which many of them have not been yet, and we will scratch the surface in this section.

To start with a well-known fact, developing a jailbreak framework is an outstanding project because many vulnerabilities must be researched and eventually a list of exploits (0-day, 1-day and N-day) should be implemented, being that in an exploit chain the breaking of only one of such exploits potentially destroys the entire chain. From this point, attacking a device remotely is equally a complicated goal because many of commercial tools try to discovery and use 0-days (0-click or not), persistence is also an issue, the invasion itself must be stealth and make the detection even harder. By the way, both these attacks are really and completely different one each other, but both are equally hard. Maybe, in a macro-visualization, a simplified and possible **sequence is searching for a remote-control execution, finding a way to escape from the sandbox, trying to compromise the kernel, finally getting persistence via an implant** and, sometimes**, cleaning up any possible tracks**. However, detailing needed steps, things are substantially more complicated because can involve, in terms of iOS, a list of tasks such as **breaking the iOS boot chain**, **exploiting a vulnerability of an application** or **daemon** (**lockdownd** daemon, which is responsible for the communication between the mobile device and the host for applications like iTunes, was a preferred one, as well as **afcd** daemon that is the **Apple File Conduite service**), tackling the code signing that, unavoidably

forces us to getting a way to circumventing or even disabling the **AMFI** (and remember that **CoreTrust** acts as a second layer of defense by mitigating fake-signing technique that is used to deploy non-authorized binaries), **getting an escape from the sandbox**, obtaining an **elevation of privilege**, discovering a **read/write kernel primitive** (potentially handling with the kernel address randomization -- **KASLR** -- and **zone allocation slide values -- harder to discover on iOS on arm64**) and, at the end, **patch some important piece of code in the kernel**.

Getting a **read-write primitive** is a constant in several operating systems, and on iOS/macOS it is not different. In general, most exploitations occur through the web browser, and there getting a read-write primitive consist of manipulating data storage such as **Typed Array** and **ArrayBuffer** to, for example, force a type-confusion vulnerability (yes, finding raw pointers have harder after the introduction of the sandbox -- check the previous article of this series: https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/).

Whatever approach that can be used, the goal is obtaining a **shellcode execution**, and that is one more difficult challenge to be overcome because read-write-execution regions cannot be allocated and also any **dynamic code execution** demands **dynamic-codesigning entitlement** (once again, entitlements show how powerful this layer of protection is). Therefore, attackers were used to allocating read-write-execute memory regions using **MAP_JIT flag** from **mmap** (just once) and even they attacked applications like WebKit to find and overwrite any jitted function code with the desired shellcode, but none of such alternatives are no longer available because there are new mitigations to prevent such tricks, which prevents attackers to find the jitted memory region because its address is randomized. Thus, any attempt to write into a read-execute region (first region) containing the function pointer is redirected to a read-write mapped region (second region) that is allocated randomly and even the original read-execution region (first region) is marked as execute-only, which was exported in previous versions and represented a point-of vulnerability, but it is not exported in recent versions of iOS and modern devices) to block any reading operation from the read-write region (second region). Apparently, the JIT region has been unified in some context and the access permission is controlled by a system register.

As I highlighted previously, **Mach IPC** plays an essential role in macOS and iOS exploitation, and details and concepts associated are really relevant to understanding tactics and tricks used by researchers over the years. We reviewed that **Mach port rights** can be sent via **Mach messages** (they are received by threads, which really execute code, and such **right is associated to a task** because we are talking about Mach), and there can be **multiple SEND rights (MACH_PORT_RIGHT_SEND and MACH_PORT_RIGHT_SEND_ONCE) associated with a port**, but **only one RECEIVE right (MACH_PORT_RIGHT_RECEIVE) associated with it**. As consequence, **SEND rights can be cloned** , but **RECEIVE ones cannot**.

We have also learned that **Mach messages** can be simple and complex, can be sent, and received through **mach_msg** or **mach_msg_overwrite** APIs, and in case of **simple Mach message** we have the following composition, even though the body is not regarded :

- **Mach message: mach_msg_base_t**
- **Mach header: mach_msg_header_t**
- **Mach body: mach_msg_body_t**

The definition of both structures is available in **osfmk\mach\message.h** file, as shown below:

```
589: typedef struct {
590:     mach_msg_size_t msgh_descriptor_count;
591: } mach_msg_body_t;
592:
593: #define MACH_MSG_BODY_NULL              ((mach_msg_body_t *) 0)
594: #define MACH_MSG_DESCRIPTOR_NULL        ((mach_msg_descriptor_t *) 0)
595:
596: typedef struct {
597:     mach_msg_bits_t              msgh_bits;
598:     mach_msg_size_t              msgh_size;
599:     mach_port_t                  msgh_remote_port;
600:     mach_port_t                  msgh_local_port;
601:     mach_port_name_t             msgh_voucher_port;
602:     mach_msg_id_t                msgh_id;
603: } mach_msg_header_t;
```

[Figure 118]: mach_msg_body_t and mach_msg_header_t structures

The fields from **mach_msg_header_t structure** have the following meaning:

- **msgh_bits:** it determines whether the message type (simple or complex), as well as the action involved and if it is a reply, for example. In general, it contains status and meta-information on the message. As this field is a bitmap, some of valid fields are **MACH_MSGH_BITS_COMPLEX** (if it is not on then the port contains a OOL data or port right, but if it is complex the value is **0x80000000**), **MACH_MSGH_BITS_REMOTE** (0x0000001f), **MACH_MSGH_BITS_LOCAL** (0x00001f00) and **MACH_MSGH_BITS_VOUCHER** (0x001f0000)
- **msgh_size:** it contains the size of the message, including header and body, and such information is quite important for the receiving side to determine how much data is being transmitted.
- **msgh_remote_port:** it contains the destination of the message (remember that we are at the Mach level). It must specify a valid send or send-once right for a port.
- **msgh_local_port:** it indicates the port that will be used as reply-port. Thus, this field carries the send-once right that the receiver will use to reply to the message.,
- **msgh_voucher_port:** this field involves a different kind of port type named voucher, which is usually used for tracking resource usage.
- **msgh_id:** this field holds the message ID, which is an identifier that is used for distinguishing the message's type.

The **message structure** is shown in the same file:

```
636: typedef struct {
637:     mach_msg_header_t            header;
638:     mach_msg_body_t              body;
639: } mach_msg_base_t;
```

[Figure 119]: mach_msg_base_t structure

As I have state previously, **if the message is simple, the message's body is completely ignored by the kernel.** If the message is **complex**, the body is not ignored, and **the given types of fields** follow the message header:

- the count of type descriptors.
- An array containing type descriptors elements (**mach_msg_descriptor_t**).
- inline data.

And, as shown previously, the available Mach message descriptors types (**mach_msg_*_descriptor_t**), given by the **mach_msg_descryptor_type_t**, are the following:

- **MACH_MSG_PORT_DESCRIPTOR:** it provides the description of a port right.
- **MACH_MSG_OOL_DESCRIPTOR:** it provides a description of a region of memory (including the respective address and size) that is not part of the message, but it is referred to by the message. If readers want to trace a rough comparison then think about an overlay in a Windows binary, which does not make part of the binary, but it is referenced by the binary.
- **MACH_MSG_OOL_PORTS_DESCRIPTOR:** it describes an array of port names, which also includes the address of the port names and respective count. It is apparently harmless to security, but as heap is allocated and used to hold the array of pointers then it can combined with heap spraying.
- **MACH_MSG_OOL_VOLATILE_DESCRIPTOR:** it points to the data to be deallocated when the message is sent.
- **MACH_MSG_GUARDED_PORT_DESCRIPTOR:** it is a descriptor for a special kind of port named guarded port, which are protected by guard mechanisms, which prevent them from being deallocated.

The **mach_msg_description_type_t** definition and the respective types are provided by **osfmk\mach\message.h** file, as shown below:

```
380: typedef unsigned int mach_msg_descriptor_type_t;
381:
382: #define MACH_MSG_PORT_DESCRIPTOR                0
383: #define MACH_MSG_OOL_DESCRIPTOR                 1
384: #define MACH_MSG_OOL_PORTS_DESCRIPTOR           2
385: #define MACH_MSG_OOL_VOLATILE_DESCRIPTOR        3
386: #define MACH_MSG_GUARDED_PORT_DESCRIPTOR        4
387:
388: #define MACH_MSG_DESCRIPTOR_MAX MACH_MSG_GUARDED_PORT_DESCRIPTOR
389:
390: #if XNU_KERNEL_PRIVATE && __has_feature(ptrauth_calls)
391: #define __ipc_desc_sign(d) \
392:     __ptrauth(ptrauth_key_process_independent_data, \
393:         1, ptrauth_string_discriminator("ipc_desc." d))
394: #else
395: #define __ipc_desc_sign(d)
396: #endif /* KERNEL */
397:
398: #pragma pack(push, 4)
399:
400: typedef struct {
401:     natural_t                   pad1;
402:     mach_msg_size_t             pad2;
403:     unsigned int                pad3 : 24;
404:     mach_msg_descriptor_type_t  type : 8;
405: } mach_msg_type_descriptor_t;
```

[Figure 120]: mach_msg_descryptor_type_t structure

OOL (Out-Of-Line transfer) is a kind of messages that carry the address of a memory region because sender and received shares the memory region, and in the past such a type has been used for heap spraying using data provided and controlled by the user.

It is an interesting moment to clear a subtle concept and that, in many opportunities, it may cause a misunderstanding. When I described **MACH_MSG_OOL_PORTS_DESCRIPTOR** above, I mentioned *"it describes an array of port names"*. The issue is that a port can be represented by a **mach_port_name_t** or a **mach_port_t** from the user's point of view (user-space), but it does not occur in the kernel-side (kernel-space). The **mach_port_name_t** is defined by the XNU source code as the local identity for a Mach Port), and it is used to identify the rights holds by the port associated with the task. In another words, **mach_port_name_t** means only a name, which can be referred to, and no rights. Therefore, **mach_port_name_t** is important inside of the namespace, and it does not have any meaning out of it. Now, from the **kernel-space**, **right** is represented by a **pointer to the IPC port object**, and there is no port namespace for the rights. Once again: **mach_port_name_t** is almost meaningless out of its namespace in the user-space. In imprecise, allusive and educational parallel, ports assume a kind of **duality port name - port right** behavior (yes, it is an implicit analogy and reference to the particle-wave dualism from **Heisenberg** in Physics -- https://en.wikipedia.org/wiki/Werner_Heisenberg), which depends on the context (user-space or kernel-space). Therefore, from the kernel point of view, once a task receives a **mach_port_t**, such a port is bound to port rights, and then the potential receiver "gaining" the referred right, which from a higher point of view is provided to application via a handle. This flux can be interpreted like **application → handle → mach_port_t → port rights → object**. If the reader considers that port rights can be sent through messages and can even be added to existing rights (check **mach_port_insert_right** function from **osfmk\ipc\mach_port.c**, which can insert a right into a space), everything becomes still more interesting:

```
1691: /*
1692:  *  Routine:    mach_port_insert_right [kernel call]
1693:  *  Purpose:
1694:  *      Inserts a right into a space, as if the space
1695:  *      voluntarily received the right in a message,
1696:  *      except that the right gets the specified name.
1697:  *  Conditions:
1698:  *      Nothing locked.
1699:  *  Returns:
1700:  *      KERN_SUCCESS        Inserted the right.
1701:  *      KERN_INVALID_TASK   The space is null.
1702:  *      KERN_INVALID_TASK   The space is dead.
1703:  *      KERN_INVALID_VALUE  The name isn't a legal name.
1704:  *      KERN_NAME_EXISTS    The name already denotes a right.
1705:  *      KERN_INVALID_VALUE  Message doesn't carry a port right.
1706:  *      KERN_INVALID_CAPABILITY Port is null or dead.
1707:  *      KERN_UREFS_OVERFLOW Urefs limit would be exceeded.
1708:  *      KERN_RIGHT_EXISTS   Space has rights under another name.
1709:  *      KERN_RESOURCE_SHORTAGE  Couldn't allocate memory.
1710:  */
1711:
1712: kern_return_t
1713: mach_port_insert_right(
1714:     ipc_space_t                 space,
1715:     mach_port_name_t            name,
1716:     ipc_port_t                  poly,
1717:     mach_msg_type_name_t        polyPoly)
1718: {
1719:     if (space == IS_NULL) {
1720:         return KERN_INVALID_TASK;
1721:     }
1722:
1723:     if (!MACH_PORT_VALID(name) ||
1724:         !MACH_MSG_TYPE_PORT_ANY_RIGHT(polyPoly)) {
1725:         return KERN_INVALID_VALUE;
1726:     }
1727:
1728:     if (!IP_VALID(poly)) {
1729:         return KERN_INVALID_CAPABILITY;
1730:     }
```

[Figure 121]: mach_port_insert_right function

There is another interesting detail that maybe are suitable for this section is that a **task port** works as a kind of special type of **Mach port**, which represents a task at the end point and, as expected, it offers the concept of ownership (own by the kernel) and identification because both concepts are regarded while managing the task, even because the **task port** is used to send and receive messages to and from the port. Actually, when I mention managing the task, I also include reading, writing, and allocating memory in the task memory region. As I have mentioned the concept of task, it would be on-time to list the task structure (**osfmk\kern\task.h**), but its definition has impressive 414 lines, which makes no viable to reproduce here. Eventually, the reader will find the field **bsd_info_ro**, which is a pointer to the **proc_ro structure** from **bsd\sys\proc_ro.h** file and whose content is:

```
60: /*!
61:  * @struct proc_ro
62:  *
63:  * @brief
64:  * Store read-only data associated to a task and/or proc
65:  *
66:  * @discussion
67:  * The lifetime of a @c proc_ro structure is 1:1 with that
68:  * of a @c proc_t or a @c task_t. @c proc_t and @c task_t
69:  * point to the same @c proc_ro, except for corpses which
70:  * have an invalid and uninitialized @c proc_t, and the
71:  * proc_data field is uninitalized.
72:  */
73: struct proc_ro {
74:     struct proc *pr_proc;
75:     struct task *pr_task;
76:
77:     __xnu_struct_group(proc_ro_data, proc_data, {
78:         uint64_t p_uniqueid;
79:         int p_idversion;
80:         uint32_t p_csflags;
81:         SMR_POINTER(struct ucred *) p_ucred;
82:         uint8_t *__unsafe_indexable syscall_filter_mask;
83:         struct proc_platform_ro_data p_platform_data;
84:     });
85:
86:     __xnu_struct_group(task_ro_data, task_data, {
87:         /* Task security and audit tokens */
88:         struct task_token_ro_data task_tokens;
89: #ifdef CONFIG_MACF
90:         struct task_filter_ro_data task_filters;
91: #endif
92:         uint32_t t_flags_ro;
93:         uint32_t task_control_port_options;
94:     });
95: } « end proc_ro » ;
96:
```

[Figure 122]: proc_ro structure from bsd\sys\proc_ro.h

We can see that there are a **proc** and **task reference** (**pr_proc* and **pr_task*) inside this structure, which seems to work as a decoupling structure between the **Mach (task)** and **BSD (process)** worlds, but that shows their bidirectional inter-connection. As there is a clear relationship between BSD and Mach, and also the translation between processes and task, there is **Mach Traps** as **task_for_pid**. Before proceeding, it is good to clear that readers can refer to a trap as a function, but it is named as **Mach trap** due traps provide application from user-space a route to communicate and request services from kernel, which is exactly the intercommunication between BSD and Mach layers through Mach messages (using **mach_msg routine**, for example, and which is also a trap). Returning to the topic and in a few words, **task_for_pid** trap (**bsd\kern\kern_proc.c**) gets a **task port** for another processes, which is referred by its **PID** on the same

host and returns a **Mach port** with send right to the referred task. In general, **receiving rights as return grants the possibility to send and receive Mach message to the message queue, and everything depends on the kind of received rights**. The **task_for_pid function (trap)** is shown below:

```
5376: kern_return_t
5377: task_for_pid(
5378:     struct task_for_pid_args *args)
5379: {
5380:     mach_port_name_t        target_tport = args->target_tport;
5381:     int                     pid = args->pid;
5382:     user_addr_t             task_addr = args->t;
5383:     proc_t                  p = PROC_NULL;
5384:     task_t                  t1 = TASK_NULL;
5385:     task_t                  task = TASK_NULL;
5386:     mach_port_name_t        tret = MACH_PORT_NULL;
5387:     ipc_port_t              tfpport = MACH_PORT_NULL;
5388:     void                    * sright = NULL;
5389:     int                     error = 0;
5390:     boolean_t               is_current_proc = FALSE;
5391:     struct proc_ident       pident = {0};
5392:
5393:     AUDIT_MACH_SYSCALL_ENTER(AUE_TASKFORPID);
5394:     AUDIT_ARG(pid, pid);
5395:     AUDIT_ARG(mach_port1, target_tport);
5396:
5397:     /* Always check if pid == 0 */
5398:     if (pid == 0) {
5399:         (void) copyout((char *)&tret, task_addr, sizeof(mach_port_name_t));
5400:         AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
5401:         return KERN_FAILURE;
5402:     }
5403:
5404:     t1 = port_name_to_task(target_tport);
5405:     if (t1 == TASK_NULL) {
5406:         (void) copyout((char *)&tret, task_addr, sizeof(mach_port_name_t));
5407:         AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
5408:         return KERN_FAILURE;
5409:     }
5410:
5411:
5412:     p = proc_find(pid);
5413:     if (p == PROC_NULL) {
5414:         error = KERN_FAILURE;
5415:         goto tfpout;
5416:     }
5417:     pident = proc_ident(p);
5418:     is_current_proc = (p == current_proc());
5419:
5420: #if CONFIG_AUDIT
5421:     AUDIT_ARG(process, p);
5422: #endif
```

[Figure 123]: task_for_pid function from bsd\kern\kern_proc.c

The **task_for_pid** function is much longer than shown above, throughout its body there are multiple interesting points, and a few of them are:

- **task_for_pid_posix_check:** it verifies that the current process should be permitted to get the task port of the target process, and this checking includes a handful of points.
- **port_name_to_task:** it brings us to *port_name_to_task_kernel*, which converts a port name to a task reference.
- **task_get_task_access_port:** it is translated to *task_get_special_port*, which calls *task_get_special_port_internal* function, and from this point, many other ones are invoked.
- **convert_task_to_port:** it calls *convert_task_to_port_kernel*, which invokes *convert_task_to_port_with_flavor* function, and it calls multiple routines such as *ipc_kobject_make_send*. From this point, a series of IPC functions.

During such introductory facts exposed until now, there are two structures that appeared as types in recent figures, but they have not been commented yet, and they are **mach_port_t** and **ipc_port_t**, which the former one is defined in **osfmk\mach\port.h file** as a single line:

- **typedef ipc_port_t mach_port_t;**

The definition above takes us to check the **ipc_port_t** definition in **osfmk\ipc\ipc_port.h** file:

```
119: struct ipc_port {
120:     struct ipc_object              ip_object;
121:     union {
122:         /*
123:          * The waitq_eventmask field is only used on the global queues.
124:          * We hence repurpose all those bits for our own use.
125:          *
126:          * Note: if too many bits are added, compilation will fail
127:          *       with errors about "negative bitfield sizes"
128:          */
129:         WAITQ_FLAGS(ip_waitq
130:             , ip_fullwaiters:1            /* Whether there are senders blocked on a full queue */
131:             , ip_sprequests:1             /* send-possible requests outstanding */
132:             , ip_spimportant:1            /* ... at least one is importance donating */
133:             , ip_impdonation:1            /* port supports importance donation */
134:             , ip_tempowner:1              /* dont give donations to current receiver */
135:             , ip_guarded:1                /* port guarded (use context value as guard) */
136:             , ip_strict_guard:1           /* Strict guarding; Prevents user manipulation of context values directly */
137:             , ip_specialreply:1           /* port is a special reply port */
138:             , ip_sync_link_state:3        /* link the port to destination port/ Workloop */
139:             , ip_sync_bootstrap_checkin:1 /* port part of sync bootstrap checkin, push on thread doing the checkin */
140:             , ip_immovable_receive:1      /* the receive right cannot be moved out of a space, until it is destroyed */
141:             , ip_immovable_send:1         /* No send(once) rights to this port can be moved out of a space, never unset */
142:             , ip_no_grant:1               /* Port wont accept complex messages containing (ool) port descriptors */
143:             , ip_tg_block_tracking:1      /* Track blocking relationship between thread groups during sync IPC */
144:             , ip_pinned:1                 /* Can't deallocate the last send right from a space while the bit is set */
145:             , ip_service_port:1           /* port is a service port */
146:             , ip_has_watchport:1          /* port has an exec watchport */
147:             , ip_kernel_iotier_override:2 /* kernel iotier override */
148:             , ip_kernel_qos_override:3    /* kernel qos override */
149:             , ip_reply_port_semantics:3   /* reply port defense in depth type */
150:             );
151:         struct waitq            ip_waitq;
152:     } « end {anon_union} » ;
153:
154:     struct ipc_mqueue               ip_messages;
155:
156:     /*
157:      * IMPORTANT: Direct access of unionized fields are highly discouraged.
158:      * Use accessor functions below and see header doc for possible states.
159:      */
160:     union {
161:         struct ipc_space       *XNU_PTRAUTH_SIGNED_PTR("ipc_port.ip_receiver") ip_receiver;
162:         struct ipc_port        *XNU_PTRAUTH_SIGNED_PTR("ipc_port.ip_destination") ip_destination;
163:         ipc_port_timestamp_t    ip_timestamp;
164:     };
165:
166:     /* update ipc_kobject_upgrade_locked() if this union is changed */
167:     union {
168:         uintptr_t               ip_kobject; /* manually PAC-ed, see ipc_kobject_get_raw() */
169:         ipc_importance_task_t   ip_imp_task; /* use accessor ip_get_imp_task() */
170:         struct ipc_port        *ip_sync_inheritor_port;
171:         struct knote           *ip_sync_inheritor_knote;
172:         struct turnstile       *ip_sync_inheritor_ts;
173:     };
174:
175:     /*
176:      * ip_specialreply:  ip_pid
177:      * ip_has_watchport: ip_twe
178:      * else:             ip_pdrequest
179:      */
180:     union {
181:         int                     ip_pid;
182:         struct task_watchport_elem *XNU_PTRAUTH_SIGNED_PTR("ipc_port.ip_twe") ip_twe;
183:         struct ipc_port *XNU_PTRAUTH_SIGNED_PTR("ipc_port.ip_pdrequest") ip_pdrequest;
184:     };
185:
186: #define IP_KOBJECT_NSREQUEST_ARMED      ((struct ipc_port *)1)
187:     struct ipc_port                *ip_nsrequest;
188:     ipc_port_request_table_t XNU_PTRAUTH_SIGNED_PTR("ipc_port.ip_request") ip_requests;
189:     struct turnstile               *ip_send_turnstile;
190:     mach_vm_address_t               ip_context;
```

```
191:
192: #if DEVELOPMENT || DEBUG
193:     natural_t                       ip_srp_lost_link : 1;   /* special reply port turnstile link chain broken */
194:     natural_t                       ip_srp_msg_sent : 1;    /* special reply port msg sent */
195:     natural_t                       ip_impcount : 30;       /* number of importance donations in nested queue */
196: #else
197:     natural_t                       ip_impcount;            /* number of importance donations in nested queue */
198: #endif
199:     mach_port_mscount_t             ip_mscount;
200:     mach_port_rights_t              ip_srights;
201:     mach_port_rights_t              ip_sorights;
202:
203:     union {
204:         ipc_kobject_label_t XNU_PTRAUTH_SIGNED_PTR("ipc_port.kolabel") ip_kolabel;
205:         /* Union of service and connection ports' message filtering metadata */
206:         void * XNU_PTRAUTH_SIGNED_PTR("ipc_port.ip_splabel") ip_splabel;
207:     };
208:
209: #if MACH_ASSERT
210:     unsigned long                   ip_timetrack;   /* give an idea of "when" created */
211:     uint32_t                        ip_made_bt;     /* stack trace (btref_t) */
212:     uint32_t                        ip_made_pid;    /* for debugging */
213: #endif  /* MACH_ASSERT */
214: } « end ipc port » ;
```

[Figure 124]: ipc_port structure

As you can see, at the beginning of the **ipc_port** definition, there is the reference to **ipc_object** from **osfmk\ipc\ipc_object.h** file:

```
109: /*
110:  * The ipc_object is used to both tag and reference count these two data
111:  * structures, and (Noto Bene!) pointers to either of these or the
112:  * ipc_object at the head of these are freely cast back and forth; hence
113:  * the ipc_object MUST BE FIRST in the ipc_common_data.
114:  *
115:  * If the RPC implementation enabled user-mode code to use kernel-level
116:  * data structures (as ours used to), this peculiar structuring would
117:  * avoid having anything in user code depend on the kernel configuration
118:  * (with which lock size varies).
119:  */
120: struct ipc_object {
121:     ipc_object_bits_t _Atomic io_bits;
122:     ipc_object_refs_t _Atomic io_references;
123: } __attribute__((aligned(8)));
```

[Figure 125]: ipc_object structure

In the image above, **ipc_object_bits_t** is a type for **natural_t**, which refers to **__darwin_natural_t** that is an **unsigned int** while the **io_references** holds references to objects. Thus, in other words, both holds are pointer and references to another object type. Furthermore, the **ipc_space structure**, used in multiple definitions, it is also quite interesting and well-commented:

```
123: struct ipc_space {
124:     lck_ticket_t    is_lock;
125:     os_ref_atomic_t is_bits;        /* holds refs, active, growing */
126:     ipc_entry_num_t is_table_hashed;/* count of hashed elements */
127:     ipc_entry_num_t is_table_free;  /* count of free elements */
128:     SMR_POINTER(ipc_entry_table_t XNU_PTRAUTH_SIGNED_PTR("ipc_space.is_table")) is_table; /* an array of entries */
129:     task_t XNU_PTRAUTH_SIGNED_PTR("ipc_space.is_task") is_task; /* associated task */
130:     thread_t        is_grower;      /* thread growing the space */
131:     ipc_label_t     is_label;       /* [private] mandatory access label */
132:     ipc_entry_num_t is_low_mod;     /* lowest modified entry during growth */
133:     ipc_entry_num_t is_high_mod;    /* highest modified entry during growth */
134:     struct bool_gen bool_gen;       /* state for boolean RNG */
135:     unsigned int    is_entropy[IS_ENTROPY_CNT]; /* pool of entropy taken from RNG */
136:     int             is_node_id;     /* HOST_LOCAL_NODE, or remote node if proxy space */
137: #if CONFIG_PROC_RESOURCE_LIMITS
138:     ipc_entry_num_t is_table_size_soft_limit; /* resource_notify is sent when the table size hits this limit */
139:     ipc_entry_num_t is_table_size_hard_limit; /* same as soft limit except the task is killed soon after data collection */
140: #endif /* CONFIG_PROC_RESOURCE_LIMITS */
141: };
```

[Figure 126]: ipc_space structure

Although I do not have enough time to show the involvement of these structures in this article, which is only the first one in a mini-series, it should be clear to readers that these structures are really associated with many exploits on macOS and iOS because such structures play a key role in **Mach layer**.

There is a quite attractive and important security aspect related to this specific **Mach trap** that I would like to comment on. The first observation is that when we have or receive send right to the **kernel task port**, we have read, write and allocation access to the memory of this process and, due to the involved and called functions from **task_for_pid** function (Mach trap) that are able to **allocate, manipulate** and **read memory**. However, as the pid==0 represents the kernel, it takes us to the second observation that we actually get access to the kernel memory address space. Obviously, we cannot simply call **task_for_pid(0)** from an unprivileged sandboxed application, and Apple has implemented foundation countermeasures such as KPP and KTRR (more about them will be shown later) against attacks that tried to call **task_for_pid** with process 0, and an error is immediately returned if such as is not performed from a privileged process or processes with the same user ID. The general idea was to get a read-write kernel primitive and patch this part of the kernel (this attack is known as **TFP0 patch**) to get unrestricted access to the kernel memory. Obviously, there are other counter-measures to prevent attacker from getting pointer to the **kernel_task** (task associated to the kernel, whose pid is equal to 0), mainly when the request does not come from the kernel or even is the own **kernel_task**, or if the **kernel_task** is got, it will be useless because import Mach APIs, which do direct communication with kernel, will refuse to accept it as argument due to multiple checks.

There are other attacks such as **Faking Task Port** and **voucher_swap**. The **Faking Task Port attack** consists in spraying the kernel heap with pointers to Mach ports (check structures that have been presented, but these pointers will be overwritten latter to point to fake Mach ports that are actually pointers to user-space heap. In this case, when the kernel uses the fake Mach port, it is really accessing a user-space pointer, which can be used for leaking information, escaping sandbox and other vulnerabilities. In case of **voucher_swap attack** (and old attack), the attack uses the **task_swap_mach_voucher function** to swap Mach vouchers, which are used for tracking and managing resources, between tasks without regarding ownership. This lack of appropriate ownership checking leads to a **user-after-free vulnerability**. About the voucher itself, check for **osfmk\ipc\ipc_voucher.h**, **osfmk\ipc\ipc_voucher.c** and **osfmk\mach\mach_voucher.defs**:

```
66: /*
67:  * IPC Voucher
68:  *
69:  * Vouchers are a reference counted immutable (once-created) set of
70:  * indexes to particular resource manager attribute values
71:  * (which themselves are reference counted).
72:  */
73: struct ipc_voucher {
74:     os_ref_atomic_t         iv_refs;        /* reference count */
75:     iv_index_t              iv_table[MACH_VOUCHER_ATTR_KEY_NUM];
76:     ipc_port_t              iv_port;        /* port representing the voucher */
77:     struct smrq_slink       iv_hash_link;   /* link on hash chain */
78: };
79:
80: #define IV_NULL         IPC_VOUCHER_NULL
81:
```

[Figure 127]: ipc_voucher definition in osfmk\ipc\ipc_voucher.h file

The **task_swap_mach_voucher**, which is generated MIG routine, can be found in **osfmk\mach\task.defs** and it is as simples as you can imagine:

```
routine task_swap_mach_voucher(
        task        : task_t;
        new_voucher : ipc_voucher_t;
    inout   old_voucher : ipc_voucher_t);
```

[Figure 128]: task_swap_mach_voucher routine

However, its definition comes from **osfmk\kern\task.c**, as shown below:

```
8613: kern_return_t
8614: task_swap_mach_voucher(
8615:     __unused task_t         task,
8616:     __unused ipc_voucher_t  new_voucher,
8617:     ipc_voucher_t          *in_out_old_voucher)
8618: {
8619:     /*
8620:      * Currently this function is only called from a MIG generated
8621:      * routine which doesn't release the reference on the voucher
8622:      * addressed by in_out_old_voucher. To avoid leaking this reference,
8623:      * a call to release it has been added here.
8624:      */
8625:     ipc_voucher_release(*in_out_old_voucher);
8626:     OS_ANALYZER_SUPPRESS("81787115") return KERN_NOT_SUPPORTED;
8627: }
```

[Figure 129]: task_swap_mach_voucher function

The note is quite funny and suitable *"To avoid leaking this reference, a call to release it has been added here."* because the exploited vulnerability comes exactly from the issue related to object reference counting (the **ipc_voucher structure** of **Figure 127** from the prior page has its first field **iv_refs** as a reference count) , which is a well-known concept from any operating system for controlling when an object should be marked to be released (count == 0) or not (count != 0) through garbage collector mechanisms. A leak of this reference count can offer the opportunity to artificially increase or decrease its value, being that once this reference is set to zero, the **voucher object** would be available to be freed even with existing pointers referring to it. The problem in the opposite direction is also important because such a reference could be overflown. Additionally, **thread_get_mach_voucher** and **thread_set_mach_voucher** functions can be used for retrieving and setting the voucher's reference to voucher from the user-space:

```
3402: /*
3403:  *  thread_get_mach_voucher - return a voucher reference for the specified thread voucher
3404:  *
3405:  *  Conditions:  nothing locked
3406:  *
3407:  *  NOTE:       At the moment, there is no distinction between the current and effective
3408:  *      vouchers because we only set them at the thread level currently.
3409:  */
3410: kern_return_t
3411: thread_get_mach_voucher(
3412:     thread_act_t            thread,
3413:     mach_voucher_selector_t __unused which,
3414:     ipc_voucher_t           *voucherp)
3415: {
3416:     ipc_voucher_t           voucher;
3417:
3418:     if (THREAD_NULL == thread) {
3419:         return KERN_INVALID_ARGUMENT;
3420:     }
```

[Figure 130]: thread_get_mach_voucher function (truncated) | osfmk\kern\thread.c

```
3438: /*
3439:  *  thread_set_mach_voucher - set a voucher reference for the specified thread voucher
3440:  *
3441:  *  Conditions: callers holds a reference on the voucher.
3442:  *      nothing locked.
3443:  *
3444:  *  We grab another reference to the voucher and bind it to the thread.
3445:  *  The old voucher reference associated with the thread is
3446:  *  discarded.
3447:  */
3448: kern_return_t
3449: thread_set_mach_voucher(
3450:     thread_t                thread,
3451:     ipc_voucher_t           voucher)
3452: {
3453:     ipc_voucher_t old_voucher;
3454:     ledger_t bankledger = NULL;
3455:     struct thread_group *banktg = NULL;
3456:     uint32_t persona_id = 0;
3457:
3458:     if (THREAD_NULL == thread) {
3459:         return KERN_INVALID_ARGUMENT;
3460:     }
3461:
3462:     bank_get_bank_ledger_thread_group_and_persona(voucher, &bankledger, &banktg, &persona_id);
3463:
3464:     thread_mtx_lock(thread);
3465:     /*
3466:      * Once the thread is started, we will look at `ith_voucher` without
3467:      * holding any lock.
3468:      *
3469:      * Setting the voucher hence can only be done by current_thread() or
3470:      * before it started. "started" flips under the thread mutex and must be
3471:      * tested under it too.
3472:      */
3473:     if (thread != current_thread() && thread->started) {
3474:         thread_mtx_unlock(thread);
3475:         return KERN_INVALID_ARGUMENT;
3476:     }
```

[Figure 131]: thread_set_mach_voucher function (truncated) | osfmk\kern\thread.c

This article, which is quite introductory, does not aim to show details, but the **Heap Feng Shui approach (as known as Port Feng Shui)** that makes part of the process of exploitation via **task_swap_mach_voucher** function is performed using **OOL Port Descriptors**, which have been quickly explained in previous sections. This type of Heap Feng Shui transmits a series of **MACH_MSG_OOL_PORTS_DESCRIPTOR** type messages (there is a **mach_msg_descriptor_type_t type** field in the **mach_msg_type_descriptor_t** structure), which makes part of the IPC mechanisms and are Mach messages (**osfmk\mach\mach_msg**), to the kernel, allocating a series of continuous blocks. Proceeding with this idea, as IPC object pointer are a kind of reference to these ports inside messages, it would be feasible to create a fake **ipc_object** in the user-space and overflow the message references point to it.

In the last couple of years, other attacks have come up, and of these clever techniques is a type of bug known as **physical use-after-free**. In general words, a process from the user-space allocates **a virtual memory region**, which is marked with **read and write permission**. Afterwards, the associated page tables of this process are updated to point out (map) the related physical address with read and write permission too. As we are talking about a use-after-free attack, the process deallocates the allocated memory from the use-space. Thus, at this moment of time, the associated pages are free. However, the trick is exactly at this point because actually is the mapping on the page table that is marked as free and eventually removed, but the real physical pages are still there and, for the kernel they would be free (and they are not). At the end of the process, the process can read and write the content of such physical pages that will

be reused by the kernel, and such resource can be read and manipulated without the kernel knowledge. From the adversary's point of view, a **heap spray** (typically allocating **IOSurface objects**) would be one of many possibilities to exploit this class of vulnerability because the attacker still has the effective control of the pages allocated from the physical memory and, once these pages are allocated, the goal is to overwrite a few of the **IOSurface object's field** to get arbitrary kernel read and write primitive.

All these attacks, counter-measures and difficulties are only a few challenges present in a wide portfolio of contexts because in modern versions of iOS the kernel space is separated from the user space (this feature is implemented by the ARM processor | **PAN -- Privileged Access Never**), which prevents execution in user space from the kernel space and also there are different kernel protections that have been implemented over time (more about them ahead).

As readers already know, in the past attackers started using **OOL (Out-Of-Lne) descriptors** to send memory address regions between both kernel and user memory regions through the classical **vm_map_copy** structure and, even though it is an old technique, it is important to mention it here. Actually, since iOS 10.x the UAF approach with **Heap Feng Shui** using **vm_map_copy structure** (below) is no longer (it is an also an old technique possible because there is a control about activity of memory releasing, so the **vm_map_copy** has been replaced by other ones . In the current days, exploiting heap vulnerabilities has become harder as reader can imagine, and most daily application have been implemented sandbox protections too, which makes the system security even more solid.

```
519: struct vm_map_copy {
520: #define VM_MAP_COPY_ENTRY_LIST          1
521: #define VM_MAP_COPY_KERNEL_BUFFER       2
522:     uint16_t                type;
523:     bool                    is_kernel_range;
524:     bool                    is_user_range;
525:     vm_map_range_id_t       orig_range;
526:     vm_object_offset_t      offset;
527:     vm_map_size_t           size;
528:     union {
529:         struct vm_map_header                hdr;    /* ENTRY_LIST */
530:         void *XNU_PTRAUTH_SIGNED_PTR("vm_map_copy.kdata") kdata;  /* KERNEL_BUFFER */
531:     } c_u;
532: };
```

[Figure 132]: vm_map_copy structure (from osfmk/vm/vm_map_xnu.h)

Eventually, one of the most well-known protections that has been introduced a long time ago is the **Kernel Patch Protection (KPP)**, which executes in **EL3 (Secure Monitor)**, and prevents the **kernel (running in EL1)** to be changed (user-mode code runs on **EL0**) because **it protects the kernel at segment level**. However, as **KPP** was not able to protect different kernel data structures, **KTRR protection** has been created, and it is actually a hardware protection used to **intercept and prevent unauthorized write tasks against read only regions.**

There are really many different attacks that could be done through an exploit or even a malware. Attackers have tried distinguished vectors such as **web browsers, applications in general, SMM (and similar formats) applications** and **Baseband (radio)**. On browsers there have been one more prolific sources of vulnerabilities, but SMM demands further attention and research, so I recommend that, if you got interested in learning more about it, read the following articles and slides:

- **The Fully Remote Attack Surface of the iPhone:** https://googleprojectzero.blogspot.com/2019/08/the-fully-remote-attack-surface-of.html
- **iMessage: malformed message bricks iPhone:** https://project-zero.issues.chromium.org/issues/42450907
- **Look, No Hands! -- The Remote, Interaction-less Attack Surface of the iPhone:** https://i.blackhat.com/USA-19/Wednesday/us-19-Silvanovich-Look-No-Hands-The-Remote-Interactionless-Attack-Surface-Of-The-iPhone.pdf
- **A Look at iMessage in iOS 14:** https://googleprojectzero.blogspot.com/2021/01/a-look-at-imessage-in-ios-14.html

Changing the purpose, a typical jailbreak operation aims to bypass the mandatory code signing of iOS and, to accomplish this goal, it changes the **Apple Mobile File Integrity daemon (amfid)**, which is exactly the responsible for checking code signatures in user mode (take a note the verification is not the integrity of the signature itself, but also the validation of certificate, potential revoked certificates and even the **provisioning profile**) . Going deeper, another way to do such bypass might be trying to add hashes into the array of trusted hashes, which is inside the kernel, without needing to disable **code-signing system**. Both methods are hard to do (nothing is really easy to do) and there are protections to prevent such attempts, but everything depends on the context of a system. Please, we also have to realize **entitlements,** which are a really granular form of permission, must be regarded because they effectively restrict what can be done or not.

Although I have provided a quick description about a few techniques and vulnerabilities, they are not even close of an endless possibilities and real attacks over years because attackers continue exploiting memory corruption of classic applications such as Safari, Mail and Messages, there are several tricks to bypass the Gatekeep, and mainly malicious documents are able to perform sandbox escaping, which as we have learned, it is one of first steps to really exploit macOS and iOS. Actually, the approaches to escape sandbox varies, and exploiting XPC services and sandboxd daemon have been one of most used, but they are not necessary the simplest, and different methods such as the already mentioned malicious documents have become more common, even though protections such as Quarantine and associated flags have to be managed.

Over the years, Apple have implemented impressive security and including many protections layers, which cover different ranges of protections, since well-known and “basic” mitigations (as well-known as user-mode/userland protections) such **ASLR (Address Space Layout Randomization)**, **DEP (Data Execution Prevention**) as known as **NX (No-Execute)**, **CFI (Control Flow Integrity)** and other ones against typical vulnerability classes up to specific protection that covers particular technologies and features. While common protections like NX/DEP can be bypassed using ROP (and the same technique can be applied to circumvent similar protection in other operating systems), there is not a standard and operational way to beat ASLR (or KASLR), and we depend on leaks (information leak or even a leak primitive) from kernel to bypass such a protection target-by-target.

As I quickly described, additional to traditional protections mentioned above, the path to get complete and full exploitation involves finding and getting working a remote-control execution (RCE), escape the **Sandboxing**, discovering and exploiting a read-write kernel vulnerability, managing to circumvent the

CoreTrust (improve the signature verification in kernel-mode and in the entire signature process involving **AMFI** as a whole), manipulating and bypassing the **AMFI** (remember that without getting **AMFI** out of the way is not possible to run arbitrary and unsigned code), and other challenges that I am not mentioning. In each of step and sub-step, there are several types of protections to prevent the attacker from making progress. At the end of the day, Apple has followed a trend of providing its operating systems (macOS and iOS) with definitive solutions to try to extinguish (or almost) entire classes of vulnerabilities (as UAF, for example), which is the same approach adopted by other players and, in a concise form, a short list of the most known protections and involved protections follow below:

A. **App sandbox** (formerly known as Seatbelt):

   This counter-measure establishes protection and limits the access of the application data and system resources. The sandboxing of an application limits system is called by the application, which strongly restricts operations that it can do. The entitlement **com.apple.security.app-sandbox**, which is built in the code signature, enables the sandboxing for the application (enforced for iOS, but not for macOS), and all user-mode process is sandboxed through the evaluation of the system-wide profile . Additionally, we should remember that macOS applications must choose to be sandboxed (the **_libsecinit_initalizer function** from **libsystem_secinit.dylib**, does the job by invoking **_libsecinit_appsandbox** which finally calls **__sandbox_ms**) while on the iOS the sandbox is enabled forcefully by the kernel, and we have as examples **WebContent, Mach traps and syscalls** that are also sandboxed as well as sandbox profiles that can be far more restrictive. Anyway, it is one of the first protections that attackers need to overcome.

B. **Pointer Authentication Code (PAC):**

   PAC is a security feature offered by the hardware, which applies unpredictable tags to pointers, and effectively prevents data and instruction pointers to be replaced with any other address or data because tags are not known in advance. The tag is a cryptographic signature, whose keys are stored into registers, and that is **applied to the unused bits of a pointer**, and such signature checked before the execution flow being transferred to it. The PAC protection uses five keys such **as IA, IB, DA, DB** and **GA keys**, where **A keys are generated at each boot** and **B keys are generated at each process creation**, and all of them are used **to sign protected kernel instructions and data**, where each user space process has an associated and unique B key, and different items are protected **function pointers (IA), function return (IB), virtual tables entries(IA), virtual table pointers (DA), kernel state (GA),** and other usages. In general, PAC has been adopted for each important data structure and object in iOS.

   With the PAC, the iOS full chain has become more complicated because it is typically necessary to circumvent the sandbox (maybe not in IOS browser), AMFI, find an arbitrary read-write primitive (as known as address-arithmetic-read and address-arithmetic-write), bypass PAC (it kills ROP/JOP), execute a shellcode, eventually perform EoP, find a kernel arbitrary read-write and finally also bypass the PAC at the kernel level. Of course, there are multiple variations to this sequence, but certainly it is never simple and now there are many additional kernel protections that have evolved from old KPP and KTRR to new ones such as PPL or SPTM/TXM. In terms of Assembly, there are

instructions that apply PAC (pac prefix), authenticate pointer (aut prefix), and other ones. As readers might have noticed, building ROP gadgets is much harder because pointers used by the chain would need to be signed. A short example is shown below:

```
om.apple.kernel:__text:FFFFFFF008709BF4 ; Attributes: bp-based frame info_from_lumina
om.apple.kernel:__text:FFFFFFF008709BF4
om.apple.kernel:__text:FFFFFFF008709BF4 ; bool __cdecl IOService::handleOpen(IOService *__hidden this, IOService *
om.apple.kernel:__text:FFFFFFF008709BF4 __ZN9IOService10handleOpenEPS_jPv       ; CODE XREF: AppleDockChannelDevic
om.apple.kernel:__text:FFFFFFF008709BF4                                        ; AFKEPKextV2::handleOpen(IOServic
om.apple.kernel:__text:FFFFFFF008709BF4
om.apple.kernel:__text:FFFFFFF008709BF4 var_10= -0x10
om.apple.kernel:__text:FFFFFFF008709BF4 var_8= -8
om.apple.kernel:__text:FFFFFFF008709BF4 var_s0=  0
om.apple.kernel:__text:FFFFFFF008709BF4 var_s8=  8
om.apple.kernel:__text:FFFFFFF008709BF4
om.apple.kernel:__text:FFFFFFF008709BF4     PACIBSP
om.apple.kernel:__text:FFFFFFF008709BF8     STP             X20, X19, [SP,#-0x10+var_10]!
om.apple.kernel:__text:FFFFFFF008709BFC     STP             X29, X30, [SP,#0x10+var_s0]
om.apple.kernel:__text:FFFFFFF008709C00     ADD             X29, SP, #0x10
om.apple.kernel:__text:FFFFFFF008709C04     MOV             X8, X2
om.apple.kernel:__text:FFFFFFF008709C08     MOV             X19, X1
om.apple.kernel:__text:FFFFFFF008709C0C     MOV             X20, X0
om.apple.kernel:__text:FFFFFFF008709C10     LDR             X2, [X0,#0x40]
om.apple.kernel:__text:FFFFFFF008709C14     CBZ             X2, loc_FFFFFFF008709C70
om.apple.kernel:__text:FFFFFFF008709C18     TBZ             W8, #0, loc_FFFFFFF008709C68
om.apple.kernel:__text:FFFFFFF008709C1C     MOV             W3, W8
om.apple.kernel:__text:FFFFFFF008709C20     LDR             X16, [X20]
om.apple.kernel:__text:FFFFFFF008709C24     MOV             X17, X20
om.apple.kernel:__text:FFFFFFF008709C28     MOVK            X17, #0xCDA1,LSL#48
om.apple.kernel:__text:FFFFFFF008709C2C     AUTDA           X16, X17
om.apple.kernel:__text:FFFFFFF008709C30     MOV             X17, #0x428
```

[Figure 133]: PAC: example code

At the first instruction, **PACIBSP**, it is established **Pointer Authentication Code** for the address, and as readers quicky will notice, such instruction, which is a focused on the instruction address using key B, is added at the beginning of most of the functions in macOS ARM64e kernel (actually the instruction is **PACIB**, and the **SP suffix** is the source register and a modifier). In this same piece of code, we find **AUTDA**, which means that **Authenticate Data address** (another PAC instruction), using key A and as expected it authenticates a data address using key A.

C. **CoreTrust:** this protection, which occurs before the **AMFI** (**AppleMobileFileIntegrity**), verifies whether the binary has a legit Apple issued certificate, which aims to protect the system against fake signing. In other words, the **AMFI** (**AppleMobileFileIntegrity**), which verifies the existence of a **CMS (Cryptographic Message Syntax)** blob by invoking **_vnode_check_signature** function, is performed after **CoreTrust** has been called to validate the blob. This protection aimed to mitigate the jailbreak techniques that modified / hooked the AMFI, but there are multiple bypassing techniques.

D. **AppleMobileFileIntegrity (AMFI):** this protection, which we have already described in detail during this article, aims to block any attempt to run unsigned code, and it is composed of a kernel extension named **AppleMobileFileIntegrity.kext** and a daemon named **amfid**. Additionally, usual operations such as debugging and task ports (a concept well-related to IPC as you have learned) are also controlled by **AMFI** (remember that debugging is enabled when SIP is disabled). Another interesting role of **AMFI** is that it checks for dynamic-codesigning entitlements for memory regions with **MAP_JIT flag** because this entitlement allows code without signing, which has been described a few pages before this point. In terms of AMFI and determined contexts, there could be much

easier to use a valid digital certificate (developers one, for example) than struggling to disable AMFI. Furthermore, it is one of the first steps to obtain persistence because there can be really hard to get it in each reboot. As expected,, the second step to reach persistence is, somehow, to get the execution of an arbitrary code (a service?!) at each reboot.

E. **Bulletproof JIT:** this protection, which was introduced in 2016, aims to prevent writing directly into the JIT region, which has RWX permission. The technique used by Apple consists in emitting a code that contains secret data, which is not readable within the process. To accomplish this goal, two virtual mappings to the same physical JIT memory are created, which one of them is only-executable and the other one is read-writable (**mach_vm_remap( )**), but the location of this writeable mapping in unknown and randomly located. Once the second JIT region mapping has been created (**mach_vm_remap( )**), a special "**memcpy function**" (**jitWriteThunkGenerator**) to write into this second JIT region is generated inside the first JIT mapping, and the memory where this special **memcpy** function has been generated is changed to only-execute. At the same way, the first JIT mapped region permission is changed to RX, and the second (writable) JIT mapped region permission is changed to RW. Finally, the address of the second (writable) JIT mapping is removed (overwritten) from memory by invoking **memset_s** function. Now the JIT region seems to be unified, and the access permission is controlled directly or indirectly by a system register.

F. **Fast Permission Restrictions (as known as APRR registers):** this protection introduces a CPU register that enables or restricts per thread permissions, allowing to remove the execute permission from memory without facing overhead of manipulating the page table or even invoking a system call. There is a brief list of observations here:

   - APRR really enforces the **W^X permission scheme**. Therefore, as the JIT region has read and execute permission, any attempt to write into it will trigger a segment faut error.
   - To write the code for the first time or to update the region later it is necessary to unlock the region for writing, and this task is performed through a special unlock function, which changes the APRR register only for the chosen thread that is responsible for copying that into the JIT region while the R and X permissions are kept.
   - The adversary would need an arbitrary execution to write the arbitrary code, which would make the problem impossible to solve. In the real world, attackers started to attack **JavaScriptCore** to induce it to emit necessary instructions.

G. **JITBox:** The effectiveness of APRR is backed by PAC. Later Apple packed both JIT region and PAC into a kind of sandbox and limited possible actions for the JIT region, so is no longer possible to "jump" outside the JIT. This protection is informally known as **JITBox** and made relatively useless to inject anything into the JIT region because the code is sandboxed there.

H. **Page Protection Layer (PPL):** it is a protection that manages **page table permission to guarantee that only PPL can alter such protected pages holding page tables and user code** and, as you will realize, it has a good association with **KTTR/RoRgn** (in later bullet). Based on this object, PPL has been designed to run with more privileges than XNU kernel and it **prevents attackers from tampering with page tables** (they are read-only only, and PPL blocks only writing operations to part

of memory pages) and executable code from processes, so **enforcing any page table modification to pass through the PPL**. Furthermore, the code signing validation is also protected. However, this protection is valid only for iOS, where all executed code must be signed.

I. **Kernel Patch Protection (KPP):** It is an enabled protection that checks **__TEXT**, structures in **__DATA segments** (sandbox, AMFI and other ones) and even **__const sections** and tries to **prevent any modification of kernel and driver code**. In other words, it checks the integrity of the kernel to ensure that it is still in its original state, which would be true in case of jailbreak operations, which change kernel permissions to RWX.

J. **KTRR/RoRgn:** this composed protection prevents any modification of the iOS kernel at runtime, so it is no longer possible to write into **const segments** or even insert new code, making harder attackers to disable protections, critical data, and memory page table. Physical memory regions are marked and locked with read-only permission (RoRgn) and even a range of physical memory that is marked (mapped) as executable is restricted in the EL1 mode (kernel-mode only), which prevents any access from user-mode (EL0). As you already have realized, attackers cannot patch important parts of the kernel and, when they could execute, such code is out of reach because it is in EL1.

K. **SPTM (Secure Page Table Monitor)** and **TXM (Trusted Execution Monitor):** these protections have been introduced in recent years and work together to protect page tables for user-space and kernel-space processes. After the A15 processor, both protections work as a replacement for **PPL**. Under certain point of view, **TXM** is one of main components responsible for performing code signature validation while **STPM** signs user pointers and maintains page tables. If readers download a recent IPSW file (**iPhone15,4_18.0_22A5282m_Restore.zip**) and uncompressed it, you are going to see a series of files, but in special one related to SPTM and other one related to TXM:

```
alexandreborges@alexandremacos Firmware % pwd
/Users/alexandreborges/research/firmware/iOS18/Firmware
alexandreborges@alexandremacos Firmware %
alexandreborges@alexandremacos Firmware % ls -lh sptm.t8120.release.im4p txm.iphoneos.release.im4p
-rw-r--r--  1 alexandreborges  staff   127K Jan  9  2007 sptm.t8120.release.im4p
-rw-r--r--  1 alexandreborges  staff   142K Jan  9  2007 txm.iphoneos.release.im4p
alexandreborges@alexandremacos Firmware %
alexandreborges@alexandremacos Firmware % disarm -I sptm.t8120.release.im4p
sptm.t8120.release.im4p: This is an IM4P... with a BVX2 payload... Uncompressed 884936 bytes
Mach-O header information:
Magic:  64-bit Mach-O
Type:   executable
CPU:    ARM64e (ARMv8.3)
Cmds:   12
Size:   1896
Flags:  0x200001 (NOUNDEFS PIE)
alexandreborges@alexandremacos Firmware %
alexandreborges@alexandremacos Firmware % disarm -I txm.iphoneos.release.im4p
txm.iphoneos.release.im4p: This is an IM4P... with a BVX2 payload... Uncompressed 393376 bytes
Mach-O header information:
Magic:  64-bit Mach-O
Type:   executable
CPU:    ARM64e (ARMv8.3)
Cmds:   11
Size:   1904
Flags:  0x200001 (NOUNDEFS PIE)
alexandreborges@alexandremacos Firmware %
```

[Figure 134]: information about SPTM and TXM files

Both binaries can be initially checked using the same **disarm** command:

```
alexandreborges@alexandremacos Firmware % disarm -L sptm.t8120.release.im4p
sptm.t8120.release.im4p: This is an IM4P... with a BVX2 payload... Uncompressed 884936 bytes
LC  0: LC_SEGMENT_64            Mem: 0xfffffff027004000-0xfffffff027014000      __TEXT
           Mem: 0xfffffff027007c74-0xfffffff02701357b    __TEXT.__cstring       (C-String Literals)
           Mem: 0xfffffff02701357b-0xfffffff0270135bb    __TEXT.__binname
           Mem: 0xfffffff0270135bc-0xfffffff027013630    __TEXT.__chain_starts
           Mem: 0xfffffff027013630-0xfffffff027014000    __TEXT.__const
LC  1: LC_SEGMENT_64            Mem: 0xfffffff027014000-0xfffffff027078000      __DATA_CONST
           Mem: 0xfffffff027014000-0xfffffff027077158    __DATA_CONST.__const
LC  2: LC_SEGMENT_64            Mem: 0xfffffff027078000-0xfffffff0270b8000      __TEXT_EXEC
           Mem: 0xfffffff027078000-0xfffffff0270b7a5c    __TEXT_EXEC.__text     (Normal)
LC  3: LC_SEGMENT_64            Mem: 0xfffffff0270b8000-0xfffffff0270bc000      __LAST
           Mem: 0xfffffff0270b8000-0xfffffff0270b8008    __LAST.__pinst
LC  4: LC_SEGMENT_64            Mem: 0xfffffff0270bc000-0xfffffff0270c8000      __DATA
           Mem: 0xfffffff0270bc000-0xfffffff0270bc018    __DATA.__auth_ptr
           Mem: 0xfffffff0270bc018-0xfffffff0270bc01e    __DATA.__data
           Mem: 0xfffffff0270bc020-0xfffffff0270c1590    __DATA.__common
           Mem: 0xfffffff0270c1590-0xfffffff0270c6ad0    __DATA.__bss
LC  5: LC_SEGMENT_64            Mem: 0xfffffff0270c8000-0xfffffff0270dc000      __BOOTDATA
           Mem: 0xfffffff0270c8000-0xfffffff0270dc000    __BOOTDATA.__data
LC  6: LC_SEGMENT_64            Mem: 0xfffffff0270dc000-0xfffffff0270e0000      __LINKEDIT
LC  7: LC_SYMTAB                Symtab: 1 entries @0xd8000(884736), Strtab is 16 bytes @0xd8010(884752
LC  8: LC_DYSYMTAB
           No local symbols
            1 external symbols at index  0
           No undefined symbols
           No TOC
           No modtab
           No Indirect symbols     No External relocations
LC  9: LC_UUID                  UUID: 1002B071-3733-3B52-8297-8265542E0832
LC 10: LC_SOURCE_VERSION        Source Version:          392.0.6.0.0
LC 11: LC_UNIXTHREAD            Entry Point:             0x80388 (Mem: 0xfffffff027084388)

alexandreborges@alexandremacos Firmware % disarm -L txm.iphoneos.release.im4p
txm.iphoneos.release.im4p: This is an IM4P... with a BVX2 payload... Uncompressed 393376 bytes
LC  0: LC_SEGMENT_64            Mem: 0xfffffff017004000-0xfffffff017010000      __TEXT
           Mem: 0xfffffff017006144-0xfffffff01700b890    __TEXT.__cstring       (C-String Literals)
           Mem: 0xfffffff01700b890-0xfffffff01700faa0    __TEXT.__const
           Mem: 0xfffffff01700faa0-0xfffffff01700fae0    __TEXT.__binname
           Mem: 0xfffffff01700fae0-0xfffffff01700fb00    __TEXT.__chain_starts
           Mem: 0xfffffff01700fb00-0xfffffff01700ffff    __TEXT.__info_plist
LC  1: LC_SEGMENT_64            Mem: 0xfffffff017010000-0xfffffff017018000      __DATA_CONST
           Mem: 0xfffffff017010000-0xfffffff017010050    __DATA_CONST.__auth_ptr
           Mem: 0xfffffff017010050-0xfffffff017017d90    __DATA_CONST.__const
LC  2: LC_SEGMENT_64            Mem: 0xfffffff017018000-0xfffffff017058000      __TEXT_EXEC
           Mem: 0xfffffff017018000-0xfffffff01705710c    __TEXT_EXEC.__text     (Normal)
LC  3: LC_SEGMENT_64            Mem: 0xfffffff017058000-0xfffffff017060000      __TEXT_BOOT_EXEC
           Mem: 0xfffffff017058000-0xfffffff01705c058    __TEXT_BOOT_EXEC.__text
           Mem: 0xfffffff01705c058-0xfffffff01705c084    __TEXT_BOOT_EXEC.__bootcode    (Normal)
LC  4: LC_SEGMENT_64            Mem: 0xfffffff017060000-0xfffffff017064000      __DATA
           Mem: 0xfffffff017060000-0xfffffff017060278    __DATA.__data
           Mem: 0xfffffff017060280-0xfffffff017060cf0    __DATA.__common
           Mem: 0xfffffff017060cf0-0xfffffff017061180    __DATA.__bss
LC  5: LC_SEGMENT_64            Mem: 0xfffffff017064000-0xfffffff017068000      __LINKEDIT
LC  6: LC_SYMTAB                Symtab: 1 entries @0x60000(393216), Strtab is 16 bytes @0x60010(393232)
LC  7: LC_DYSYMTAB
           No local symbols
            1 external symbols at index  0
           No undefined symbols
           No TOC
           No modtab
           No Indirect symbols     No External relocations
LC  8: LC_UUID                  UUID: D7C2884B-4B11-3E48-9B6E-6E3C9F1B5A73
LC  9: LC_SOURCE_VERSION        Source Version:          135.0.3.0.0
LC 10: LC_UNIXTHREAD            Entry Point:             0x58000 (Mem: 0xfffffff01705c000)
alexandreborges@alexandremacos Firmware %
```

[Figure 135]: information about SPTM and TXM files -- part 2

L. Multiple other protections:

a. **PGZ (Probabilistic Guard Zalloc):** double mapping allocation of a page (as a kind of "shadow memory") that returns the secondary address to the caller. In the XNU kernel is marked as "Probabilistic gzalloc", and its description is given as:

```
659: #if CONFIG_PROB_GZALLOC
660: /*
661:  * Probabilistic gzalloc
662:  * =====================
663:  *
664:  *
665:  * Probabilistic guard zalloc samples allocations and will protect them by
666:  * double-mapping the page holding them and returning the secondary virtual
667:  * address to its callers.
668:  *
669:  * Its data structures are lazily allocated if the `pgz` or `pgz1` boot-args
670:  * are set.
671:  *
672:  *
673:  * Unlike GZalloc, PGZ uses a fixed amount of memory, and is compatible with
674:  * most zalloc/kalloc features:
675:  * - zone_require is functional
676:  * - zone caching or zone tagging is compatible
677:  * - non-blocking allocation work (they will always return NULL with gzalloc).
678:  *
679:  * PGZ limitations:
680:  * - VA sequestering isn't respected, as the slots (which are in limited
681:  *   quantity) will be reused for any type, however the PGZ quarantine
682:  *   somewhat mitigates the impact.
683:  * - zones with elements larger than a page cannot be protected.
684:  *
```

[Figure 136]: Probabilistic guard zalloc description (osfmk\kern\zalloc.c)

b. **kalloc_type:** allocations over different zones are segregated by type, so killing UAF and type confusion.
c. **kalloc_ro:** this allocator commits, as expected, read-only memory that only can be accessed if it is authorized by PPL, so helping to prevent kernel write attacks based on privilege escalation.
d. **Lockdown mode:** it disables JIT, so killing different attack, block most message attachments, incoming communication invitations are blocked, installing new profiles is not allowed, there is not interaction with any external device.
e. **mach_msg2:** the functionality is similar to standard **mach_msg** (send and receive messages between tasks), but this time binding its operation to **Control Flow Integrity** (**CFI**) that is implemented and associated with PAC, for example. Readers will see two important references on XNU, which are **mach_msg2_trap** (**osfmk\ipc\mach_msg.c**) and **mach_msg2_internal** (**libsyscall\mach\mach_msg.c**). The first one in associated with the waiting behavior of a task for a message when a task calls it (the task blocks and needs to wait until receiving the message) and second one (**mach_msg2_internal**) is the real function that receives the message, validates it, copies data and finally wakes the task up running in waiting state (via **mach_msg2_trap**). In terms of code, we can see the **mach_msg2_trap** invokes **mach_msg2_internal** at the beginning of the function, as shown in the following two figures:

```
924: /*
925:  *  Routine:     mach_msg2_trap [mach trap]
926:  *  Purpose:
927:  *      Modern mach_msg_trap() with vector message and CFI support.
928:  *  Conditions:
929:  *      Nothing locked.
930:  *  Returns:
931:  *      All of mach_msg_send and mach_msg_receive error codes.
932:  */
933: mach_msg_return_t
934: mach_msg2_trap(
935:     struct mach_msg2_trap_args *args)
936: {
937:     /* packed arguments, LO_BITS_and_HI_BITS */
938:     uint64_t mb_ss = args->msgh_bits_and_send_size;
939:     uint64_t mr_lp = args->msgh_remote_and_local_port;
940:     uint64_t mv_id = args->msgh_voucher_and_id;
941:     uint64_t dc_rn = args->desc_count_and_rcv_name;
942:     uint64_t rs_pr = args->rcv_size_and_priority;
943:
944:     mach_msg_timeout_t  msg_timeout = (mach_msg_timeout_t)args->timeout;
945:     mach_msg_vector_t   msg_vec = {}, aux_vec = {};
946:     mach_msg_size_t     vec_snd_count = 0;
947:     mach_msg_size_t     vec_rcv_count = 0;
948:     mach_msg_option64_t option64;
949:     mach_msg_return_t   mr = MACH_MSG_SUCCESS;
950:
951:     option64 = ipc_current_user_policy(current_task(),
952:         args->options) | MACH64_MACH_MSG2;
953:
954:     KDBG(MACHDBG_CODE(DBG_MACH_IPC, MACH_IPC_KMSG_INFO) | DBG_FUNC_START);
```

[Figure 137]: first lines of mach_msg2_trap (osfmk\ipc\mach_msg.c)

```
91: #if defined(__LP64__) || defined(__arm64__)
92: mach_msg_return_t
93: mach_msg2_internal(
94:     void *data,
95:     mach_msg_option64_t option64,
96:     uint64_t msgh_bits_and_send_size,
97:     uint64_t msgh_remote_and_local_port,
98:     uint64_t msgh_voucher_and_id,
99:     uint64_t desc_count_and_rcv_name,
100:    uint64_t rcv_size_and_priority,
101:    uint64_t timeout)
102: {
103:    mach_msg_return_t mr;
104:
105:
106:    mr = mach_msg2_trap(data,
107:        option64 & ~LIBMACH_OPTIONS64,
108:        msgh_bits_and_send_size,
109:        msgh_remote_and_local_port,
110:        msgh_voucher_and_id,
111:        desc_count_and_rcv_name,
112:        rcv_size_and_priority,
113:        timeout);
114:
115:
116:    if (mr == MACH_MSG_SUCCESS) {
117:        return MACH_MSG_SUCCESS;
118:    }
```

[Figure 138]: first lines of the mach_msg2_internal (libsyscall\mach\mach_msg.c)

There are other less-known protections, but these ones are enough to prove how Apple has dedicated a huge effort to keep iOS and macOS really secure.

11. References

A brief list of recommended references follows below:

- **Apple security documentation 1:** https://developer.apple.com/documentation/security
- **Apple security documentation 2:** https://security.apple.com/
- **Programming with Objective-C:** https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html
- **Apple Silicon:** https://developer.apple.com/documentation/apple-silicon
- **Arm Assembly language:** https://developer.arm.com/documentation/107829/0200/Assembly-language-basics
- **Arm Architecture Reference Manual for A-profile architecture:** https://developer.arm.com/documentation/ddi0487/latest/
- **Disabling and Enabling System Integrity Protection:** https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection
- **Generating a Non-Maskable Interrupt:** https://developer.apple.com/documentation/kernel/generating_a_non-maskable_interrupt
- **Understanding and Debugging Kernel Panics:** https://developer.apple.com/library/archive/technotes/tn2063/_index.html
- **Debugging a Custom Kernel Extension:** https://developer.apple.com/documentation/apple-silicon/debugging-a-custom-kernel-extension
- **Building and Debugging Kernels:** https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/build/build.html
- **Kernel Debug Kit for macOS – Read Me** (found inside the KDK: /Library/Developer/KDKs/KDK_<version>.kdk)
- **Method Swizzling:** https://nshipster.com/method-swizzling/
- **Libmalloc source files:** https://github.com/apple-oss-distributions/libmalloc
- **Memory Usage Performance Guidelines:** https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/ManagingMemory.html#//apple_ref/doc/uid/10000160-SW1
- **MacOS memory allocator (libmalloc) Exploitation:** https://www.slideshare.net/AngelBoy1/macos-memory-allocator-libmalloc-exploitation
- **CFMessagePort:** https://developer.apple.com/documentation/corefoundation/cfmessageport-rs2
- **NSPasteboard:** https://developer.apple.com/documentation/appkit/nspasteboard
- **Auditing and Exploiting Apple IPC:** http://thecyberwire.com/events/docs/IanBeer_JSS_Slides.pdf
- **References about code signing:** https://developer.apple.com/documentation/security/code-signing-services and https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.
- **GateKeeper:** https://support.apple.com/guide/security/gatekeeper-and-runtime-protection-sec5599b66df/web

- **Apple Notarization:** https://developer.apple.com/documentation/security/notarizing-macos-software-before-distribution
- **Hardened Runtime:** https://developer.apple.com/documentation/Security/hardened-runtime
- **A step-by-step guide to writing an iOS kernel exploit:** https://alfiecg.uk/2024/09/24/Kernel-exploit.html
- **Operating system integrity:** https://support.apple.com/en-me/guide/security/sec8b776536b/1/web/1#secd022396fb
- **iOS 17: New Version, New Acronyms:** https://www.df-f.com/blog/ios17
- **iOS 17: New Version, New Acronyms | Round 2**: https://www.df-f.com/blog/ios-17round2
- **KTRR**: https://blog.siguza.net/KTRR/
- **Examining Pointer Authentication on the iPhone XS:** https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html
- **APRR:** https://blog.siguza.net/APRR/
- **The core of Apple is PPL: Breaking the XNU kernel's kernel:** https://googleprojectzero.blogspot.com/2020/07/the-core-of-apple-is-ppl-breaking-xnu.html
- **Life and death of an iOS attacker by Luca Todesco (video):** https://www.youtube.com/watch?v=8mQAYeozl5I
- **Mac OS X Internals:** vol. 1 and 2 (authored by Amit Singh)
- **MacOS and iOS Internals:** vol. 1/2/3 (authored by Jonathan Levin)

12. Conclusion

This article presented a basic and superficial introduction to macOS/iOS internals, which is a quite extensive topic, and certainly there are vast possibilities to be explored in terms of security and mainly vulnerability research, which is our goal here.

I hope I have enough time in the near future to write a second article about the topic, where I will explain in detail some of major areas of iOS/macOS, but this time focusing on security and vulnerability aspects.

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep learning, reversing, and exploiting everything, and I will see you next time!

Alexandre Borges