VB2021
localhost

7 - 8 October, 2021 / vblocalhost.com

ARM'D & DANGEROUS
AN INTRODUCTION TO ANALYSING ARM64 MALWARE TARGETING MACOS

Patrick Wardle
Objective-See, USA

patrick@objective-see.com

ABSTRACT

Apple's new M1 systems offer a myriad of benefits ... for both *macOS* users and, apparently, malware authors too!

In this paper we detail the first malicious programs compiled to natively target *Apple Silicon* (M1/arm64), focusing on methods of analysis.

We'll start with a few foundation topics, such as methods of identifying native M1 code (which will aid us when hunting for M1 malware), as well as some introductory arm64 reversing concepts.

With an uncovered corpus of malware compiled to run natively on M1, we'll spend the remainder of the paper demonstrating effective analysis techniques, including many specific to the analysis arm64 code on *macOS*.

Armed with this information and analysis techniques, you'll be a proficient *macOS* M1 malware analyst!

INTRODUCTION

As the popularity of *macOS* continues to sky rocket, so too does the prevalence of malware targeting *Apple*'s desktop OS.

Though the reasons for this lock-step increase are rather nuanced, it's undeniable that more *macOS* systems simply means more targets. Malware authors are an opportunistic bunch, and as such, have dedicated ever more time and resources towards crafting malware capable of infecting *macOS* systems. So much so that (by some metrics), *Macs* outpaced *Windows* in terms of the number of threats detected per endpoint [1]:

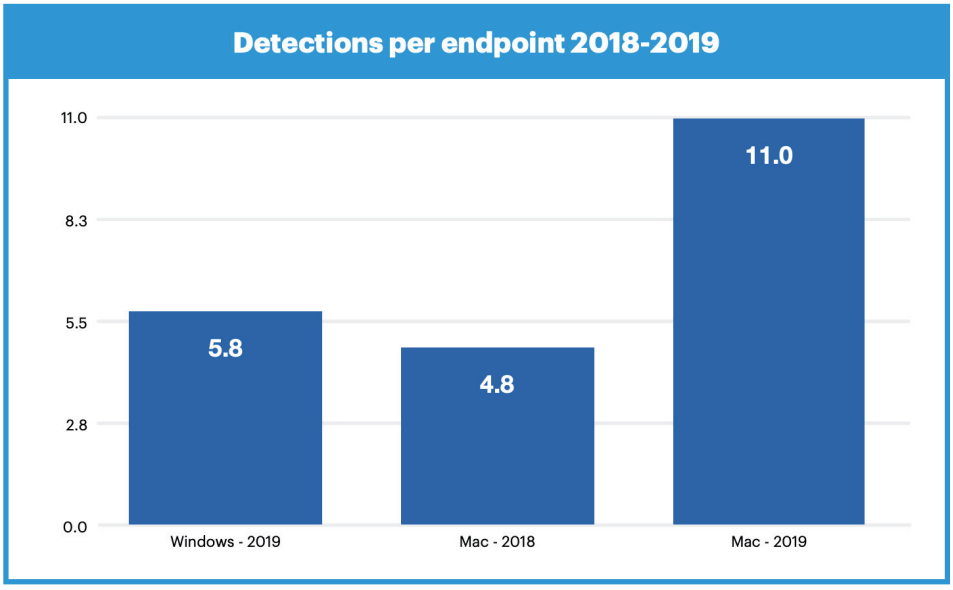

Figure 1: Detections per endpoint (source: Malwarebytes [1]).

Interesting, though somewhat unsurprising, is the fact that many recent examples of malware capable of infecting *macOS* are not wholly new. Instead, driven by the increased prevalence of *macOS*, malware authors have simply ported over their *Windows* (or *Linux*) malware. Recent examples include malware such as Dacls, IPStorm, and GravityRAT. All now run natively on *macOS*.

Of course *Mac*-specific malware also continues to abound.

We've noted that, arguably, the driving factor of the increase in *Mac* malware is the increase in *Mac* systems. And the number of such systems has increased massively in recent years, including an industry-leading 49% increase in Q4 2020 [2].

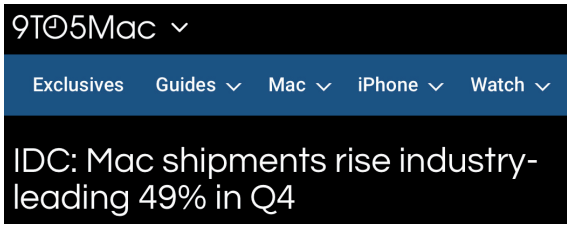

Figure 2: A significant increase in Mac shipments [2].

The reasons for *Mac*'s increased popularity can be explained by factors such as greater acceptance in the enterprise, an ever increasing remote workforce, and last but not least the introduction of *Apple*'s incredible M1 chip [3].

Mac Sales Skyrocketing After M1 Launch

Tuesday January 19, 2021 2:00 am PST by Hartley Charlton

Apple's worldwide Mac shipments grew massively in the fourth quarter of 2020 after the launch of three new Macs with the M1 chip, according to new PC shipping estimates shared by Gartner. Apple shipped an estimated 6.9 million Macs, up from the 5.25 million it shipped at the same time in 2019, marking significant growth of 31.3 percent.

Figure 3: M1 chip drives increased Mac adoption [3].

Let's briefly dive into the latter, as it's directly relevant to the core of this research paper.

Released in 2020, *Apple*'s M1 (a.k.a. '*Apple Silicon*') is 'an ARM-based system on a chip' [4].

Apple notes that:

> 'As a system on a chip (SoC), M1 combines numerous powerful technologies into a single chip, and features a unified memory architecture for dramatically improved performance and efficiency.' [5]

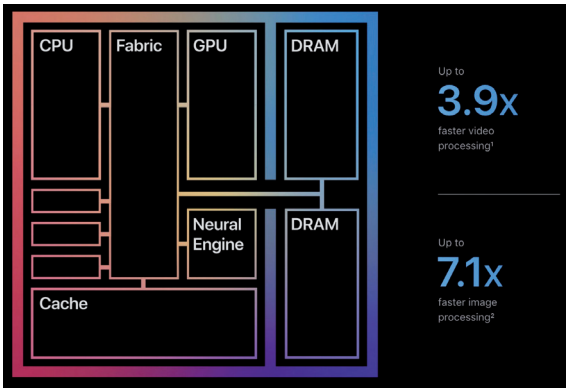

Figure 4: Apple M1 chip (source: MacRumors/Apple).

Due to the combination of its computational power and efficiency (largely due to its integrated design and ARM-based CPU instruction set), it has understandably proved massively popular with consumers and enterprise users alike.

In the context of this research paper, the most notable aspect of the M1 is that it's an ARM-based SoC, with the CPU supporting the arm64 (AArch64) instruction set architecture (ISA). Thus, in order for a binary to run natively on an M1 system, it must be compiled as a Mach-O 64-bit arm64 binary …which means developers must (re)compile their applications.

'But wait!', you might say, 'I know I can still run many older applications on my shiny new M1 system!'. And you are correct. *Apple* (wisely) realized that backwards compatibility was essential to ensure widespread customer adoption of their new M1 *Mac* systems, and thus released Rosetta(2).

As *Apple* notes:

> 'Rosetta is a translation process that allows users to run apps that contain x86_64 instructions on Apple silicon.
>
> 'To the user, Rosetta is mostly transparent. If an executable contains only Intel instructions, macOS automatically launches Rosetta and begins the translation process. When translation finishes, the system launches the translated executable in place of the original. However, the translation process takes time, so users might perceive that translated apps launch or run more slowly at times.' [6]

As summarized in the quotation above, Rosetta(2) will translate x86_64 (*Intel*) instructions transparently into native arm64 instructions, so older applications can run (almost) seamlessly on M1 systems.

However, there are two points to note:

1. Non-arm64 code will not run natively on M1 systems (the CPU only 'speaks' arm64) – it has to be translated first, via Rosetta(2).
2. As arm64 code does not have to be translated, applications (re)compiled for M1 will run natively, and thus faster, and won't be subject to any of the issues or nuances of Rosetta.

Based on the fact that native (arm64) applications run faster (as they avoid the need for runtime translation), and that Rosetta at the time of its release had a few bugs (that may prevent certain older apps from running), developers are wise to (re)compile their applications for M1!

```
Process:               oahd-helper [36752]
Path:                  /Library/Apple/*/oahd-helper
Identifier:            oahd-helper
Version:               203.13.2
Code Type:             ARM-64 (Native)
Parent Process:        oahd [506]
Responsible:           oahd [506]
User ID:               441

Date/Time:             2021-02-12 10:34:15.107 -1000
OS Version:            macOS 11.1 (20C69)

Crashed Thread:        0  Dispatch queue: com.apple.main-thread
```

Figure 5: Lost in translation: a Rosetta(2) crash.

Due to the benefits of native M1 code, it is no surprise that both developers and malware authors are shipping binaries compiled natively to run on *Apple Silicon*.

> **Note:**
> *We use several terms somewhat interchangeably in the remainder of this paper, including M1, Apple Silicon and arm64, when referring to malware compiled to run natively on this new architecture.*
>
> *Technically, as noted, M1 and Apple Silicon refer to the SoC, while arm64 is the instruction set supported by the CPU component (of the SoC).*

AND WHY THIS ALL MATTERS (TO MALWARE ANALYSTS)

Shortly, we'll discuss the discovery of the first malware compiled to natively target *Apple Silicon*. This confirmed that malicious adversaries are indeed crafting multi-architecture applications, so their code will run natively on M1 systems.

The creation of such malicious software is notable for two main reasons. First (and unsurprisingly), this illustrates that malicious code continues to evolve in direct response to both hardware and software changes coming out of Cupertino. There are a myriad of benefits to distributing native arm64 binaries, so why would malware authors resist?

Secondly, and more worryingly, (static) analysis tools or anti-virus engines may struggle with arm64 binaries. In a simple experiment, we separated out the x86_64 and arm64 binaries from a malicious application (that was compiled as a universal binary, meaning it contained multiple architecture-specific binaries).

Both the (now separated) binaries were then uploaded to *VirusTotal* and scanned. In theory, both binaries should be detected at the same rate, as they both contain the same logically equivalent malicious code.

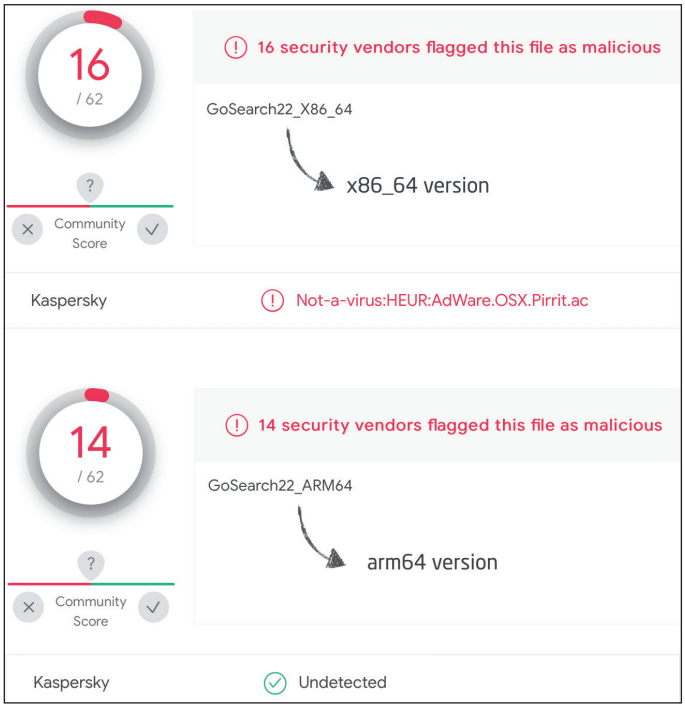

Figure 6: Detections dropped for an arm64 version.

Unfortunately, detections of the arm64 version dropped over 10% when compared to the standalone x86_64 version. Several industry-leading AV engines (that readily detected the x86_64 version) failed to flag the malicious arm64 binary.

It is surmised that, in this case, the detection signatures were based on the *Intel*-specific instructions (opcodes). As the ARM-based malware has completely different instructions, any signature based on architecture-specific instructions may fail.

Moreover, some of the AV engines that (correctly) flagged both the x86_64 and arm64 binaries as malicious presented differing names for their detections of what was logically the same file.

One AV engine with conflicting names was *Microsoft*, which named the architecture-specific files *Trojan:MacOS/Bitrep.B* and *Trojan:Script/Wacatac.C!ml*. Such naming conflicts may indicate inconsistencies when processing the differing binary file formats. Such conflicts may lead to confusion in malware identifications and reporting, which could have various real-world consequences.

Finally, though it's likely that malware compiled to run natively on *Apple Silicon* will be distributed as universal binaries (meaning they'll ship with *Intel* code as well) for the time being, this won't always be the case. At some point in the future, for example once M1 systems are more prevalent, we'll come across *macOS* malware solely containing arm64 code.

As malware analysts the presence of only arm64 code may present some challenges – most notably the fact that it disassembles not into the familiar *Intel*-based instructions, but rather ARM (arm64). The good news, though, is that armed (ha!) with a foundational understanding of this instruction set, we'll be back in business, and analysing M1 malware will be a breeze.

ANALYSING M1 BINARIES

In this section of the paper, we'll discuss various topics related to the analysis of malware designed to run natively on *Apple Silicon*.

First, how does one ascertain if a binary contains code capable of running natively on an M1 system? Well, for starters it will contain arm64 code. One simple way to determine the code contained in a binary is via *macOS*'s built-in **file** tool (the **otool** and **lipo** utilities can be used as well). Using this tool, we can examine a binary to see if it contains compiled arm64 code.

Let's look at *Objective-See*'s firewall, *LuLu*:

```
% file LuLu.app/Contents/MacOS/LuLu
Mach-O universal binary with 2 architectures:
 [arm64:Mach-O 64-bit executable arm64]
 [x86_64:Mach-O 64-bit executable x86_64]
```

Listing 1: Using the file utility to examine a universal binary.

As *LuLu* has been rebuilt to run natively on M1 systems, we can see it contains arm64 code ('Mach-O 64-bit executable arm64').

In order to maintain compatibility with older, non-M1 systems, *LuLu* also contains native *Intel* (x86_64) code.

Note that a Mach-O binary contains code and data for one architecture only. In order to create a single binary that can execute on systems with different architectures (e.g. *Intel* 64-bit and *Apple Silicon* arm64), developers can wrap multiple Mach-O binaries in a universal, or fat, binary.

When a universal binary is run, the operating system automatically selects the architecture compatible with the host. For example, when *LuLu* is run on a 64-bit *Intel* system, the x86_64 Mach-O version of the binary (which, remember is embedded directly within the universal binary) is run. On an M1 system, the arm64 Mach-O binary is executed.

HUNTING FOR MALICIOUS M1 BINARIES

When *Apple* released the M1 chip (in late 2020), it seemed reasonable to assume malware authors would shortly (re)compile their malicious creations to be natively compatible with this new architecture.

In early February 2021, I decided to hunt for such malware as none had yet been discovered or publicly disclosed.

Since I'm an independent security researcher, I don't have access to private or proprietary malware collections or feeds. Luckily, *VirusTotal* (www.virustotal.com) is generous enough to offer (free) researcher accounts. So my search began there.

VirusTotal supports a wide range of search modifiers, which are essential when hunting through the vast collection for malicious native M1 code. Such search modifiers allow one to constrain search queries by binary type, architecture(s), and much more.

To search for binaries natively compatible with *Apple Silicon*, I leverage the following search modifiers:

- macho (type): The file is a Mach-O (*Apple*) executable.
- arm (tag): The file contains ARM instructions.
- 64bits (tag): The file contains 64-bit code (recall *Apple Silicon* supports arm64).
- multi-arch (tag): The file contains support for multiple architectures (i.e. it's a universal/fat binary). As M1 systems are not yet widespread, it was assumed that malware targeting M1 would be universally compiled, to also retain native compatibility with *Intel*-based systems.

Unfortunately, constructing a search query with these search modifiers will also return (many) universal *iOS* binaries. As I was hunting solely for *macOS* binaries, I also leveraged the 'engines' search modifier along with 'iOS' and a NOT (to invert the logic).

My initial search therefore was: **type:macho tag:arm tag:64bits tag:multi-arch NOT engines:IOS**

The good news was that this did identify universal *macOS* binaries in *VirusTotal*'s corpus that contained arm64 code. The bad news was that it detected over 72,000 files.

As I was hunting for (any) M1 malware, I took a shortcut and added a search modifier that constrained the query to only detect files that have been flagged as malicious by at least two anti-virus engines.

Since I was searching for universal binaries (based on the assumption that attackers would want their malicious creations to also run on existing *Intel*-based *Apple* hardware), it seemed reasonable to expect that current AV signatures may detect at least the *Intel*-based code. And, yes, this meant the query would miss brand new (currently undetected) malware, but my goal was simply to find *any* malicious software capable of running natively on *Apple Silicon*.

The search query therefore became: **type:macho tag:arm tag:64bits tag:multi-arch NOT engines:IOS positives:2+**

This returned only 72 candidate files.

Figure 7: Candidate macOS arm64 malware.

Though several were false positives or misclassified *iOS* binaries, I quickly come across a promising candidate: a file named GoSearch22 (SHA-256: b94e5666d0afc1fa49923c7a7faaa664f51f0581ec0192a08218d68fb079f3cf).

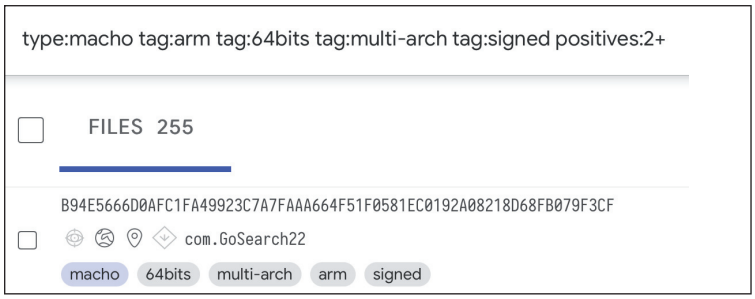

Figure 8: A likely macOS arm64 malware specimen.

Could this be the first publicly known instance of malicious code compiled to run natively on *Apple Silicon*?

TRIAGING GOSEARCH22

First, let's confirm that this is indeed a *macOS* (vs. *iOS*) binary, which can run natively on M1 systems:

```
% $ file GoSearch
Mach-O universal binary with 2 architectures:
 [arm64:Mach-O 64-bit executable arm64]

 [x86_64:Mach-O 64-bit executable x86_64]
```

Listing 2: Getting confirmation.

So far it looks good! It's a universal (fat) Mach-O binary, that supports both *Intel* (x86_64) and *Apple Silicon* (arm64).

And is it malicious? Over 20 trusted AV engines (correctly) thought so! They largely identified it as an instance of the prevalent, yet rather insidious, 'Pirrit' adware.

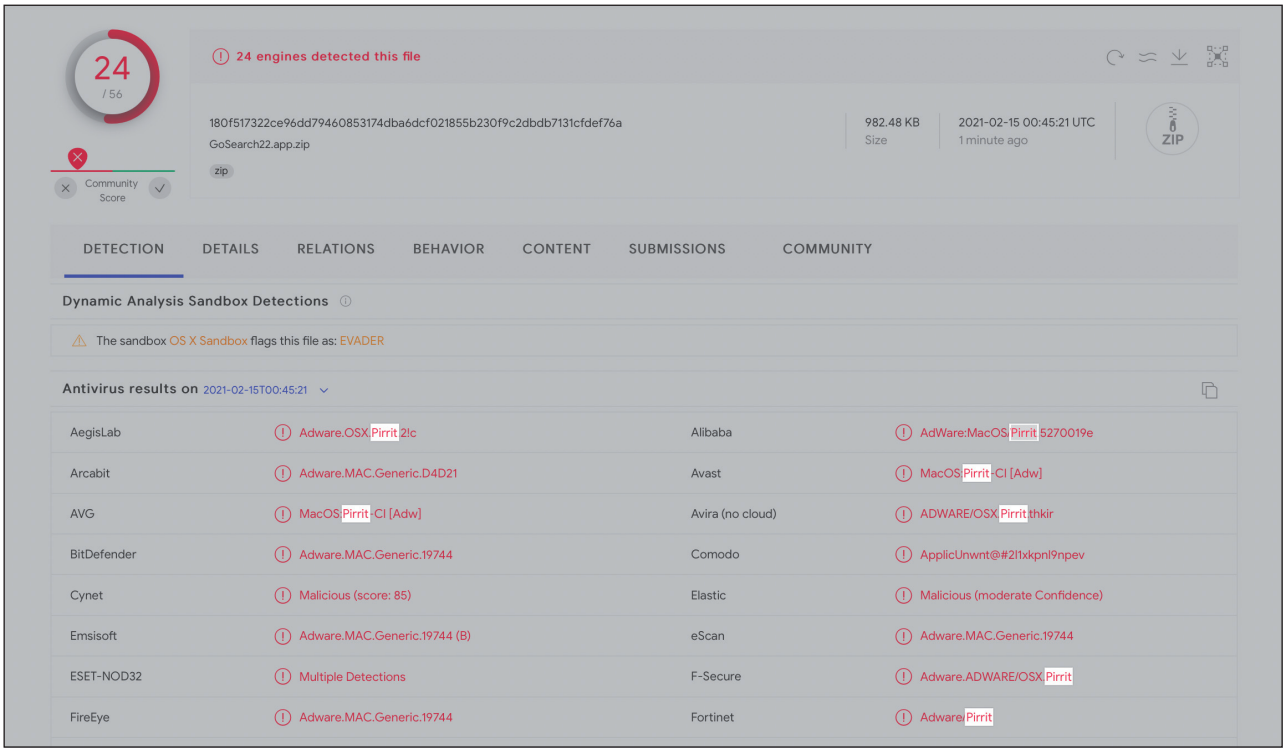

Figure 9: AV engines largely identified the file as an instance of the 'Pirrit' adware.

Moreover, at the time of uncovering this sample, *Apple* had (already) revoked its signature, indicating that they thought it was malicious as well:

Figure 10: A revoked certificate.

But wait, how is this malware 'new' if it's so well known (by over 20 AV companies)? Valid question! What is new is the fact that this malicious sample contains an embedded Mach-O binary compiled to run natively on *Apple Silicon* (M1). All previous malware, including this specimen, only contained *Intel* code, and thus would not run natively on M1 hardware.

However, it is still readily detectable for two reasons:

1. As a universal binary, it contains an *Intel* (x86_64) Mach-O binary, that the AV industry has (likely) already seen. As such, those signatures likely hit, and thus flag the entire universal binary as malicious.
2. The new arm64 binary is functionally equivalent to the *Intel* version, and thus old signatures developed to detect the *Intel* version have (some) success against the new M1 binary.

Note:

We noted earlier in this paper that detections of the arm64 version were 10% lower than the Intel version. Thus (some) existing AV signatures clearly need to be updated.

I had set out to find the first instance of malware designed to run natively on *Apple Silicon*. And hooray(?), I was able to successfully uncover such a specimen.

Now, let's detail exactly how to analyse such binaries.

A (BRIEF) INTRODUCTION TO REVERSING APPLE SILICON BINARIES

Before diving into an analysis of the M1 version of the GoSearch22 binary, we need to discuss some foundational arm64 reversing concepts. Here, we discuss registers, and various instructions of AArch64. It should be noted that this discussion is far from comprehensive, and focuses on information relevant to reversing (malicious) arm64 binaries (for example, we don't discuss floating point or SIMD concepts).

Note:

This section of the paper leans heavily upon existing arm64 reversing tutorials and papers including:

- *arm64 Assembly Crash Course [7]*
- *How to Read ARM64 Assembly Language [8]*
- *Introduction To Arm Assembly Basics [9]*
- *Modern Arm Assembly Language Programming [10]*

The reader interested in gaining a more comprehensive understanding of the topic should consult these sources.

Of course ARM's official document can always be consulted, as it is the definitive source:

- *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile [11]*

By definition, Mach-O binaries are 'binary', meaning that, while readily readable by computers, their compiled binary code is not designed to be directly readable by humans.

As such, we must leverage a tool that can understand the compiled binary machine-level code, and translate it back into something more readable: assembly code! This process is known as disassembling.

Assembly is a low-level programming language that is translated directly to binary instructions. This direct translation means that any binary code within a compiled binary can (later) be directly converted back into assembly. This process can occur irrespective of the programming language used by the developer (or malware author).

Note:

There are various versions of assembly, each compatible with various CPUs (Intel, MIPs, ARM, etc.).

Since this paper is focused on analysing arm64 macOS malware, we'll focus on AArch64, the 64-bit execution state of ARMv8 instruction set architecture (ISA). This is the native ISA of the M1 chip.

REGISTERS

Registers are temporary storage 'slots' on the CPU that can referenced by name. In some sense, they are synonymous to variables in higher-level programming languages [8].

AArch64 includes 31 64-bit registers that can be utilized for 'general purposes'. These registers are named X0 – X30 (though their lower 32 bits can be referred to via W0 – W30). It should be noted that several of these general registers have specific uses, especially in the context of function calls. Referred to as the 'calling convention', this is discussed below, as it is very relevant in the context of analysing arm64 binaries.

Besides the 31 general purpose registers, AArch64 also includes a stack pointer register (SP), and a program counter register (PC). The former points to the 'top' of the stack (though the stack grows downwards in memory). The latter, PC, contains the address of the instructi on that is currently executing [10]. Finally, a virtual register named XZR is interpreted as the value 0 [12].

CALLING CONVENTION

When a function (or method) call is made, there are strict rules (articulated in an Application Binary Interface (ABI)) that govern how registers may be utilized. For example, which registers are used to pass parameters, and which are used to return a value from the function.

As these rules are applied consistently, it allows us as malware analysts to understand exactly how a call is being made. For example, for a method that takes a single parameter, the value of this parameter (the argument) will always be passed in via the X0 register prior to the call.

- Registers X0 – X7: first eight arguments (more? … via the stack)
- Register X0/X1: return value (64, 128 bits)
- Register X29: frame pointer (to support stack frames)
- Register X30: link register (contains the return address)

(Source: [7].)

PROCESSOR STATE (PSTATE)

Unlike other ISAs there is not a dedicated flags or status register. Instead, AArch64 includes an 'abstract entity', called the PSTATE, that contains, amongst other things, conditions flags for negative, zero, carry and overflow. As we will see below, various instructions can (indirectly) update these flags, or leverage them in conditional branches [10].

AARCH64 INSTRUCTIONS

Assembly instructions map to a specific sequence of bytes that instructs the CPU to perform an operation [10]. Such instructions begin with a mnemonic, which is a human-readable abbreviation of the operation that the instructions perform. For example, the **add** mnemonic maps to the binary code to perform, you guessed it, an addition operation. The majority of arm instructions also contain one or more operands. These operands specify either the registers, values, or memory that the instruction uses [10]. Sticking with our **add** instruction, it generally takes three operands including the destination register, source register(s), and/or an immediate.

Operands can be classified into three types:

- Immediate: A constant value (e.g. 42)
- Register: One of the 31 general purpose registers
- Memory: One of the 31 general purpose registers that points to a value in memory.

Generally, the first register is the destination register, while the remaining registers are source. One notable exception is the store instruction (str, discussed below).

Let's now look at examples of the aforementioned operand types, (still) sticking with the **add** instruction. First, a simple example that makes use of both a register and an immediate operand:

```
add x0, x1, 42
```

When executed, this instruction will cause the CPU to add the immediate value 42 to the value in the x1 register. The result will be saved in the x0 register. If you're familiar with a higher-level language such as C, this will be analogous to the statement: x0 = x1 + 42;

It should be noted that one might see 'variations' on instruction mnemonics depending on nuances of the instruction. For example, the humble **mov** (move) instruction has several variants, including:

- **movz (move + zero)**: move a 16-bit value into a register, while all other bits (in the register) are set to zero.
- **movk (move + keep)**: move a 16-bit value into a register at a specified offset, while leaving the other bits in the register unmodified.
- **movn (move + negated)**: move a negated (optionally shifted) 16-bit value into a register.

Though it may help to memorize all of these, often (in the context of reversing malware) it suffices simply to understand that they are variations of the **mov** instruction. If more details are needed, the ARM instruction guide can always be consulted!

Before we discuss memory operands, we need to introduce memory address modes.

Unlike some other architectures, (such as *Intel*), Arm utilizes what is referred to as a 'load-store' architecture which separates instructions into those that can access memory and those that cannot. The former includes only the **ldr** (load register) and **str** (store register) instructions, while all others fall into the category of the latter.

> 'ARM uses a load-store model for memory access which means that only load/store (LDR and STR) instructions can access memory.
>
> …on ARM data must be moved from memory into registers before being operated on' [13].

As noted in [13], this means that if a value in memory is to be incremented, this requires:

1. Loading the value from memory into a register (via the **ldr** instruction).
2. Incrementing the value (in the register).
3. Storing the value from the register, back into memory (via the **str** instruction).

As its name implies, the **ldr** instruction loads a value from memory into a register. The following example shows how to load a 64-bit value from the memory address pointed to by the x0 register into the x1 register:

```
ldr x1, [x0]
```

Note that, as normal, the first operand (the x1 register) is the destination register. The source register is x0. The brackets around it indicate that it contains a memory address that should be dereferenced (to load the value). In C, an analogous statement would be x1 = *x0;

On the other hand, the **str** (store register) will store a value from a register into specified memory address. This instruction is somewhat unique, as the *first* operand is the source register, while the second is the destination. (Recall that for other instructions this is flipped: the first is destination, and remaining are the source(s).)

The following example shows how to store a 64-bit value from the x1 register into the memory address pointed to by the x0 register:

```
str x1, [x0]
```

Again, the brackets indicate that the register (x0) contains a memory address, which is where the value of x1 should be stored. In C, an analogous statement would be *x0 = x1;

While we are on the topic of memory accesses, we should note that there are several different 'addressing modes' that can be leveraged by the load and store instructions. We've already seen the 'base register' address mode, in the above **ldr** and **str** examples. This mode uses a single (base) register that contains the memory address. In both the previous load and store examples the xO register was the base.

Building on this, an offset can also be specified, either via an immediate or another register, such as in the following example(s):

```
ldr x1, [x0, 42]
ldr x1, [x0, x2]
```

In each example, a value is loaded from memory into the x1 register. Also in both, the x0 register is the base register, which contains the (base) memory address. In the first example, the immediate value of 42 is added as an offset to the base register (x0) before the memory is dereferenced and its value loaded into x1. In the second example, instead of an immediate value, a register value (x2) is the offset added to the base register.

We should note that there are other, more complex addressing schemes referred to as 'pre-indexed' and 'post-indexed', which modify the base register either before (pre) or after (post) its dereference. For a detailed and illustrative discussion of these addressing modes, see [13].

Finally, the 'PC relative' addressing mode can be used to compute memory addresses relative (plus or minus) to the instruction pointer (PC).

Before we wrap up the discussion of the load and store instructions, it is important to discuss their variations. For example, to load or store a 16-bit value, you'll see the **ldrh** or **srth** instructions used respectively. Also, variations exist for sign extensions. For example, the **ldrsw** load instruction will load and sign-extend a 32-bit value (into a 64-bit register).

CONDITIONS

When studying malware understanding comparison logic is often quite important. For example, malware may invoke a function that contains logic to ascertain if it's executing within a virtual machine or under the watchful eye of a debugger. Often such functions will return a boolean value (YES/NO). A comparison may then be performed on such a result, which may trigger logic such as a premature exit. Such premature exits are not conducive to analysis, and thus must be identified (so that they can be overcome).

In arm64, we noted that the processor state (PSTATE) 'register' contains various conditional flags. Various instructions can (indirectly) set these flags, such as the cmp (**compare**) instruction. 'Variations' of other instructions can do so as well. For example, the **adds** (addition, setting flags) will perform an addition *and* also updates the PSTATE flags. [10]

Once the flags have been set, subsequent instructions can act upon them using condition codes [7]. Such condition codes include **eq** (equal), **ne** (not equal), **lt** (less than), **gt** (greater than), and are often found as suffixes on instructions. For example, the **b.gt** instruction will branch (jump) if the result of a previous instruction (such as a **cmp**), is greater than.

Let's now look a bit closer at branch instructions.

BRANCHES

Branch instructions can alter the control flow of a program. The **b** instruction will instruct the CPU to unconditionally transfer control to a specified label or address (relative to the PC). As we just saw, it can also be used with a conditional code, which will only cause the transfer of control if the condition is met.

In your reversing adventures, you might also encounter the **cbz** (compare branch zero) or **cbnz** (compare branch not zero) instructions, which simply combine a compare and a conditional branch instruction.

Finally, the **bl** (branch with link) instruction will 'copy the address of the next instruction into LR', the link register (x30) [14]. This instruction facilitates function calls. When said function wants to return it simply invokes the **ret** (return) instruction. This branches (jumps) to the address that was stored in the link register (by the **bl** instruction).

REVERSING HELLOWORLD

Armed with an elementary understanding of arm64, let's now reverse a quintessential 'Hello World' binary. Specifically, the one generated by *Apple*'s Xcode:

```
int main(int argc, const char * argv[]) {
  @autoreleasepool {
    // insert code here...
    NSLog(@"Hello, World!");
  }
   return 0;
}
```

Listing 3: Hello World!

First, we compile this code via Xcode, or directly via clang (clang main.m -fmodules -o helloWorld).

Opening it in a disassembler (I use *Hopper* [15]) generates the following disassembly:

```
main:
sub    sp, sp, #0x30
stp    x29, x30, [sp, #0x20]
add    x29, sp, #0x20
movz    w8, #0x0
stur   wzr, [x29, var_4]
stur   w0, [x29, var_8]
str    x1, [sp, #0x20 + var_10]
str    w8, [sp, #0x20 + var_14]
bl     objc_autoreleasePoolPush
adrp   x9, #0x0000000100004000
add    x9, x9, #0x8 ; 0x100004008@PAGEOFF, @"Hello, World!"
str    x0, [sp, #0x20 + var_20]
mov    x0, x9
bl     NSLog
ldr    x0, [sp, #0x20 + var_20]
bl     objc_autoreleasePoolPop
ldr    w0, [sp, #0x20 + var_14]
ldp    x29, x30, [sp, #0x20]
add    sp, sp, #0x30
ret
```

Listing 4: Hello World! disassembled.

Let's triage this code, discussing relevant instructions.

First, a function prologue, where the code subtracts 0x30 from the stack pointer to make local space for the function. Then, via the **stp** instruction, it saves the x29 and x30 registers on the stack.

A few instructions later, the code invokes the objc_autoreleasePoolPush function by means of the **bl** (branch with link) instruction. (The @autoreleasepool in the source code gets compiled into a call to objc_autoreleasePoolPush and later objc_autoreleasePoolPop). Recall that before control is transferred via a **bl** instruction the link register (x30) is updated with the next instruction, so the function knows where to return. The objc_autoreleasePoolPush returns a pointer to a pool

object that must be passed to the objc_autoreleasePoolPop function. Recall that x0 contains the return value of a function, the instruction str x0, [sp, #0x20 + var_20], therefore stores (saves) this returned pointer into a local variable.

Next, the code initializes the x0 register, with the address of the 'Hello World!' string. This is accomplished by first calculating the address of the string (via the **adrp** (address of a relative page) and **add** instructions), then moving the address into the x0 register. (Recall that when a function call is made, the x0 register holds the first argument.) The NSLog function is then invoked via the **bl** instruction, to print out 'Hello World!'.

After this call, the code invokes the objc_autoreleasePoolPop function to exit the autorelease pool. As the objc_autoreleasePoolPop function takes a pool object (to release), this must be moved into the x0 register. This is accomplished via the ldr x0, [sp, #0x20 + var_20] instruction, which loads it from the stack (where it was previously stored).

The main function's prologue is then executed. This initializes the return register w0 (as the main function returns a 32-bit integer), via the value from the stack. Looking back in the disassembly, we see that the variable was initialized with zero (movz w8, #0x0, str w8, [sp, #0x20 + var_14]). Once the return register has been set, the function restores the x29 and x30 registers and (re)adjusts the stack. Finally, the **ret** instruction is executed to return.

Hooray, we've just reverse-engineered a full arm64 (*Apple Silicon*) binary!

PRACTICAL M1 MALWARE ANALYSIS

If you were slightly daunted by the previous section, that's OK. In the context of malware analysis, more often than not you won't have analyse the malware instruction by instruction.

Via dynamic analysis tools and decompiler logic, only in rare cases will you have to dive into the actual disassembly.

Case in point, *Hopper*'s decompiler can reconstruct the Hello World! source from the compiled (arm64) binary, rather impressively:

```
int main(int arg0, int arg1) {
    var_20 = objc_autoreleasePoolPush();
    NSLog(@"Hello, World!");
    objc_autoreleasePoolPop(var_20);
    return 0x0;
}
```

Listing 5: Hello World! decompiled.

The takeaway from the output of the decompiler should not be 'I don't have to learn arm64', but rather that often a fundamental understanding of arm64 will suffice.

Still, when analysing more complex malware (that contains anti-analysis logic and obfuscations), analysing the disassembly instructions may be the only option.

It is often trivial, via dynamic analysis tools, to fairly comprehensively analyse a malicious sample. For example, say you're interested in determining how a malware specimen persists. Often, you can simply execute it in conjunction with a process and file monitor. Such monitors will often quickly reveal exactly how the malware persistently installs itself.

Of course, malware authors are aware of such analysis tactics and thus may implement anti-analysis logic to thwart our analysis efforts. Such logic often seeks to ascertain if the malware is running on an analysis machine and/or is being debugged. If the malware determines that it is likely being watched, it will prematurely exit or simply idle.

Thus, as malware analysts, we must both identify and then circumvent any such anti-analysis logic so that analysis can continue. Often this requires an in-depth analysis of the malware disassembled code.

As an illustrative example, let's take a dive into GoSearch22, specifically focusing on its anti-analysis logic.

GoSearch22 is an example of malicious code that implements various anti-analysis logic. If we run it in a virtual machine or within a debugger it simply terminates. This hinders our abilities to understand how it persists and the capabilities of its payload.

But not to worry! Armed with our foundational knowledge of arm64, let's dig into the sample with the goal of uncovering such anti-analysis logic. Such an exercise will allow us to apply (and expand) our arm64 reversing skills as well as provide a means to overcome GoSearch22's anti-analysis logic so our analysis can continue.

> ***Note:***
>
> *Many of GoSearch22's anti-analysis techniques can be found in other (unrelated) malware samples. Thus gaining an understanding of such anti-analysis techniques will prove useful even when analysing other malicious code.*

> ***Note:***
>
> *Dynamic analysis of potentially malicious binaries should always be done within an isolated virtual machine, or better yet, on a separate analysis machine. We opt for the latter, largely because virtualization of M1 binaries is not yet supported (for example in VMware Fusion).*

GOSEARCH22: ANTI-DEBUGGING LOGIC

When executing GoSearch22 in a debugger (lldb), it quickly terminates:

```
% lldb GoSearch22.app
(lldb) target create "GoSearch22.app"
Current executable set to '/Users/user/Downloads/GoSearch22.app' (arm64).

(lldb) c
Process 654 resuming
Process 654 exited with status = 45 (0x0000002d)
```

Listing 6: GoSearch22 exits when executed in a debugger.

The exit code, 45 (0x2d), is telling. Experienced *Mac* malware analysts will recognize this status code as the results of the debuggee invoking the ptrace system call (or API), with the PT_DENY_ATTACH flag. As its name implies the PT_DENY_ATTACH flag instructs the operating system to prevent the debuggee from being debugged.

Malware, of course, would rather not be debugged, so it's unsurprising that GoSearch22 implements such anti-analysis logic. It is rather trivial to bypass this anti-analysis technique in a debugger simply by skipping over the ptrace call. Of course, this requires first locating where the malware invokes ptrace.

Looking at GoSearch22's disassembly reveals massive numbers of junk instructions aimed at complicating static analysis (such as locating anti-analysis logic). Moreover, there appears to be a call to the user-mode ptrace API. Thus, we assume the malware is instead making a direct call to the ptrace system call (number 0x1a). The arm64 assembly instruction to make a system call is **svc** (supervisor cal).

Searching through the disassembly for an **svc** instruction with a parameter of ptrace (0x1a), we find the responsible anti-debugging code at 0x00000001000541e8:

```
0x00000001000541e8 movz    x0, #0x1a
0x00000001000541ec movz    x1, #0x1f
0x00000001000541f0 movz    x2, #0x0
0x00000001000541f4 movz    x3, #0x0
0x00000001000541f8 movz    x16, #0x0
0x00000001000541fc svc     #0x80
0x0000000100054200 movz    w11, #0x6b8f
```

Listing 7: Anti-debugging logic (via a ptrace system call).

First, the x0 register is initialized with 0x1a, the system call number for ptrace (SYS_ptrace). The x1 register is set to 0x1f, the value of PT_DENY_ATTACH. The other two arguments, x2 and x3, are set to zero. Then at 0x00000001000541fc, the supervisor call is made. As mentioned earlier, this attempts to prevent debugging or, if the malware is being debugged, will cause the malware to terminate with exit code 45 (0x2d).

Now we've detected the location of the anti-debugging logic, in our debugging session we can simply skip the call. How? By setting a breakpoint on the **svc** instruction, and then once hit, changing the address of the program counter (pc register) to the next instruction:

```
(lldb) reg write $pc 0x100054200
```

Listing 8: Modifying the program counter register, to skip the problematic ptrace call.

As the **svc** instruction is skipped, it will not be executed, resulting in the anti-debugging logic being avoided!

…but wait, unfortunately there is more.

Even with the anti-debugging check bypassed, if the malware is allowed to continue execution (in the debugger), it still terminates prematurely. Turns out there is more anti-debugging logic.

Spread amongst the junk instructions, we find a call to the **sysctl** API:

```
0x0000000100054fcc ldur     x8, [x29, var_B8]
0x0000000100054fd0 movz     w9, #0x288
0x0000000100054fd4 str      x9, [x8]
0x0000000100054fd8 ldur     x0, [x29, var_C8]
0x0000000100054fdc ldur     x3, [x29, var_B8]
0x0000000100054fe0 ldur     x2, [x29, var_A8]
0x0000000100054fe4 orr      w1, wzr, #0x4
0x0000000100054fe8 movz     x4, #0x0
0x0000000100054fec movz     x5, #0x0
0x0000000100054ff0 bl       sysctl
```

Listing 9: Anti-debugging logic (via a sysctl).

The **sysctl** API can be invoked in order to retrieve various information, including details about the state of the current process. Such details include a flag that will be set if the program is being debugged. This is illustrated in the following C code:

```
int name[4];
struct kinfo_proc processInfo;
size_t size = sizeof(processInfo);

name[0] = CTL_KERN;
name[1] = KERN_PROC;
name[2] = KERN_PROC_PID;
name[3] = getpid();

sysctl(name, 4, &processInfo, &size, NULL, 0);

if(0 != (info.kp_proc.p_flag & P_TRACED))
{
       //debugger detected
}
```

Listing 10: PoC anti-debugging logic (via a sysctl).

In the arm64 code the sysctl API is invoked at 0x0000000100054ff0 via the **bl** (branch with link). The two previous instructions initialize the fifth and sixth arguments to zero. Continuing backwards, at 0x0000000100054fe4, the second argument is set to 4. As this argument is a 32-bit integer, the w1 register (the 32-bit part of the x1 register) is used. It is set to zero by bitwise OR'ing the 32-bit zero register with 4.

The first, third and fourth (x0, x2, x3) arguments are all initialized via the **ldur** (load unscaled register) instruction.

The first argument (x0) is a pointer to the 'name' array. In a debugger we can print out its values (via the x/4wx command):

```
(lldb) x/4wx $x0
0x16fe86de0: 0x00000001 0x0000000e 0x00000001 0x00000475
```

Listing 11: Displaying the name array passed to the sysctl API.

The values correspond to CTL_KERN (0x1), KERN_PROC, (0xe), KERN_PROC_PID (0x1), and the current process id of the malware.

The third argument (x2) is an out pointer to a kinfo_proc structure. Once the sysctl function is executed it will contain the requested details: the information about the currently running process.

Finally, the fourth argument (x3) is initialized with the size of the kinfo_proc structure, or 0x288. This initialization takes four instructions:

```
0x0000000100054fcc ldur     x8, [x29, var_B8]
0x0000000100054fd0 movz     w9, #0x288
0x0000000100054fd4 str      x9, [x8]
0x0000000100054fd8 ldur     x0, [x29, var_C8]
...
0x0000000100054fdc ldur     x3, [x29, var_B8]
```

Listing 12: The kinfo_proc structure initialization.

First, the **ldur** instruction loads the address of the size variable (var_B8) in the x8 register. Then the size of the kinfo_proc structure (0x288) is moved into the w9 register via the **movz** instruction. The **str** (store) instruction then stores this value (in x9) into the address stored in the x8 register. Finally, this value is loaded into the x3 register via the **ldur** instruction, to complete the argument initialization.

After the **sysctl** call is made, the malware examines the now populated kinfo_proc structure. Specifically, it checks if the p_flag flag has the P_TRACED bit set. If this bit is set, the malware knows it's being debugged and will (prematurely) exit.

The following instructions extract the p_flag member from the populated kinfo_proc structure (whose address was copied into the 'var_90' variable):

```
0x000000010005478c  ldur    x8, [x29, var_90]
0x0000000100054790  ldr     w8, [x8, #0x20]
0x0000000100054794  stur    w8, [x29, var_88]
```

Listing 13: The p_flag extraction from the kinfo_proc structure.

First, the address of the kinfo_proc structure is loaded into the x8 register (via the **ldur** instruction). Then the 32-bit p_flag member, which is found at offset 0x20 within the structure, is loaded into the w8 register (via the **ldr** instruction). This value is then stored in the var_88 variable via the **stur** (store unscaled register) command.

Later, the malware checks if the p_flags flag has the P_TRACED bit set (P_TRACED is the constant 0x00000800, meaning it has the 11th bit set to 0x1). In a debugging session, we can confirm that indeed, as expected, the p_flags flag has the P_TRACED bit set:

```
(lldb) p/t $w8
0b00000000000000000101100000000110
```

Listing 14: Confirming the p_flags is indeed set.

Here are the arm64 instructions that are executed to extract the P_TRACED bit:

```
0x0000000100055428  ldur   w8, [x29, var_88]
0x000000010005542c  ubfx   w8, w8, #0xb, #0x1
0x0000000100055430  sturb  w8, [x29, var_81]
```

Listing 15: The P_TRACED bit extraction from the p_flag member of the kinfo_proc structure.

In the previous instructions, the malware first loads the saved p_flag value (var_88) into the w8 register via the **ldur** instruction. Then it executes the **ubfx** (unsigned bit field extract) instruction to extract the P_TRACED bit. The **ubfx** instruction takes a destination register (w8), a source register (w8), the bitfield index (0xb, or 11d), and the width (1, for a single bit). In other words, it's grabbing the bitfield at offset 11 from the p_flag. This is the P_TRACED bit. Via the **sturb** (store unscaled register byte) instruction, it then saves the extracted P_TRACED bit. Later, it checks (compares) to make sure the P_TRACE bit is not set:

```
0x00000001000550ac ldurb  w8, [x29, var_81]
0x00000001000550b0 cmp    w8, #0x0
...
```

Listing 16: Checking the extracted P_TRACED bit.

If the P_TRACED bit is set, the malware (prematurely) exits, as this indicates the malware is being debugged.

To bypass this second anti-debugging check, we can (once again) just skip the problematic call. Specially, once the malware is about to execute the branch instruction to invoke sysctl, we can change the program counter to the next instruction. As the sysctl call is not made, the kinfo_proc structure remains uninitialized (with zeros), meaning any checks on the P_TRACED flag will return 0 (false).

The final two anti-analysis checks the malware performs involve checking if it is running within a virtual machine or if SIP is disabled. Both are likely indicators of an analysis environment. For example, malware analysts often disable SIP on an analysis system to facilitate debugging and memory inspection.

Both these checks are performed via the shell. Specifically, the malware executes various shell commands and parses their output. If either check fails, the malware prematurely exits which may thwart our analysis efforts.

For example, the 'am I running in a VM?' check involves executing the following, which looks for various artifacts found within a visual machine:

```
/bin/sh -c -c,

readonly VM_LIST="VirtualBox\|Oracle\|VMware\|Parallels\|qemu";is_hwmodel_vm()
{ ! sysctl -n hw.model|grep "Mac">/dev/null;};is_ram_vm(){((($(sysctl -n hw.memsize)/
1073741824))<4));};is_ped_vm(){ local -r ped=$(ioreg -rd1 -c IOPlatformExpertDevice);echo
"${ped}"|grep -e "board-id" -e "product-name" -e "model"|grep -qi "${VM_LIST}"||echo
"${ped}"|grep "manufacturer"|grep -v "Apple">/dev/null;};is_vendor_name_vm(){ ioreg -l|grep
-e "Manufacturer" -e "Vendor Name"|grep -qi "${VM_LIST}";};is_hw_data_vm(){ system_profiler
SPHardwareDataType 2>&1 /dev/null|grep -e "Model Identifier"|grep -qi "${VM_LIST}";};is_vm()
{ is_hwmodel_vm||is_ram_vm||is_ped_vm||is_vendor_name_vm||is_hw_data_vm;};main(){ is_vm&&echo
1||echo 0;};main "${@}
```

Listing 17: An anti-VM check.

As we're performing analysis on native hardware, this doesn't impact our analysis. On the other hand, the SIP check does impact our analysis (as SIP was disabled on the analysis machine). Let's take a closer at this check – both to understand the logic, but more importantly as a great illustrative example of reversing (and understanding) an Objective-C method call.

In order to check if the system upon which it is executing has SIP disabled, the malware executes the following via the shell, /bin/sh:

```
-c command -v csrutil > /dev/null && csrutil status | grep -v "enabled" > /dev/null &&
echo 1 || echo 0
```

Listing 18: An 'is SIP enabled' check.

If SIP is enabled, this will echo 0, whereas if it is disabled the command will echo 1.

The malware executes this command at 0x00000001000538dc, via a **blr** (branch with link to register):

```
0x00000001000538d0   ldr    x8, [sp, #0x190 + var_120]
0x00000001000538d4   ldr    x0, [sp, #0x190 + var_100]
0x00000001000538d8   ldr    x1, [sp, #0x190 + var_F8]
0x00000001000538dc   blr    x8
```

Listing 19: Execution of the SIP-detection logic.

The branch destination is held in the x8 register. Prior to the call, various parameters are prepared via the **ldr** instruction.

Due to the malware's use of obfuscation, it is not readily apparent from static analysis what address the x8 register points to. However, as we've thwarted the malware's anti-debugging logic, we can trivially ascertain this via a debugger.

```
(lldb) x/i $pc
->  0x1000538dc: 0xd63f0100   blr   x8

(lldb) reg read $x8
x8 = 0x0000000193a5f160  libobjc.A.dylib`objc_msgSend
```

Listing 20: Leveraging a debugger to determine a branch target.

Ah, it's a call to the objc_msgSend function. In short, when you (or a malware author) invokes an Objective-C method call, the compiler will route it through the objc_msgSend function.

As detailed in *Apple*'s developer documentation, the first argument is an object that the method is invoked upon. The second argument is the name of the method. Then, any arguments that the method takes:

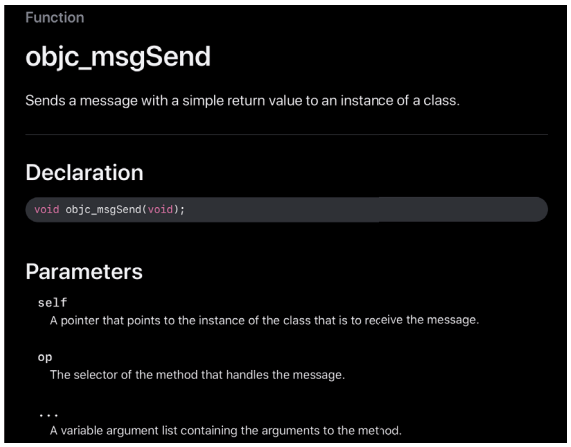

Figure 11: objc_msgSend documentation .

In our debugging session we can examine the value of these arguments to determine the object, method, and any arguments.

```
(lldb) po $x0
<NSConcreteTask: 0x1058306c0>

(lldb) x/s $x1
0x1e9fd4fae: "launch"
```

Listing 21: Leveraging a debugger to determine arguments.

First, we use the print object (po) debugger command to print out the object. It's an instance of an NSConcreteTask. To determine the method being invoked, we print out the string (x/s) that's in the second argument. It's the launch method, which, as its name implies, will launch (execute) a task.

Due to the introspective nature of Objective-C, we can query the task object, for example to extract the path of its command and any arguments. Recall that the task object is in the x0 register, as it's the first parameter for the objc_msgSend function.

```
(lldb) po [$x0 launchPath]
/bin/sh

(lldb) po [$x0 arguments]
<__NSArrayI 0x10580dfd0>(
-c,
command -v csrutil > /dev/null && csrutil status | grep -v "enabled" > /dev/null && echo
1 || echo 0
)
```

Listing 22: Leveraging a debugger to introspect an NSTask object.

To summarize, the malware is performing a 'is SIP disabled?' check as a means to determine if it's likely running in an analysis environment. This is accomplished by invoking the launch method of the NSTask (NSConcreteTask) class, which gets routed through the objc_msgSend function. By introspecting the task's launch path and arguments we can uncover the specific command (and command arguments) that the malware was executing to perform this anti-analysis check. With this information, we can also trivially side-step this anti-analysis logic (for example by skipping over the method call).

This wraps up the malware's anti-analysis logic, which, once identified, is trivial to bypass and allows continued analysis to commence! Such continued analysis is beyond the scope of this paper, largely as traditional (read: non-arm64 specific) dynamic analysis techniques suffice. For example, via tools such as file and process monitor, one can observe the malware attempting to install itself as a malicious *Safari* extension. Such an extension aims to subvert users' browsing sessions by engaging in traditional adware-type behaviours.

Note:

Since our discovery of GoSearch22, a handful of other native arm64 malware has been discovered and analysed. These include, but are not limited to:

- *Clipping Silver Sparrow's wings: Outing macOS malware before it takes flight [16]*
- *OSX/Hydromac: New Mac adware, leaked from a flashcards app [17]*

Moreover, re-running our query on VirusTotal (type:macho tag:arm tag:64bits tag:multi-arch NOT engines:IOS positives:2+) reveals a myriad of new malicious specimens, natively compiled for Apple Silicon. The interested reader should investigate :)

CONCLUSION

Macs continue to surge in popularity, in part driven by the introduction of the impressive M1 chip. By uncovering malicious code built to run natively on this ARM-based architecture, we confirmed that malware authors have been quick to adapt. And thus so too must we.

As malware built to run natively on M1 systems will disassemble to arm64, it's imperative to possess an understanding of this instruction set. This paper sought to provide such an understanding though an introduction to arm64 and its instruction set. Moreover, by reverse-engineering the first natively compatible M1 malware, we have provided a practical example of analysing arm64 disassembly.

Armed with a solid comprehension of the topics presented in this paper, you're now well on the way to becoming a proficient analyst of arm64 malware targeting *macOS*!

REFERENCES

[1] Malwarebytes. 2020 State of Malware Report. February 2020. https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report-1.pdf.

[2] Potuck, M. IDC: Mac shipments rise industry-leading 49% in Q4, Apple boosts market share to 8%. 9TO5Mac. January 2021. https://9to5mac.com/2021/01/11/mac-huge-q4-growth-49-percent/.

[3] Charlton, H. Mac Sales Skyrocketing After M1 Launch. MacRumors. January 2019. https://www.macrumors.com/2021/01/19/mac-sales-skyrocketing-after-m1-launch/.

[4] Wikipedia. Apple M1. https://en.wikipedia.org/wiki/Apple_M1.

[5] Apple. Apple unleashes M1. November 2020. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[6] Apple Developer. About the Rosetta Translation Environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

[7] arm64 Assembly Crash Course. https://github.com/Siguza/ios-resources/blob/master/bits/arm64.md.

[8] Wolchok, S. How to Read ARM64 Assembly Language. March 2021. https://wolchok.org/posts/how-to-read-arm64-assembly-language/.

[9] Azeria. Introduction To Arm Assembly Basics. https://azeria-labs.com/writing-arm-assembly-part-1/.

[10] Kusswurm, D. Modern Arm Assembly Language Programming. https://www.apress.com/gp/book/9781484262665.

[11] Arm Developer. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest/.

[12] hoakley. Code in ARM Assembly: Registers explained. The Electic Light Company. June 2021. https://eclecticlight.co/2021/06/16/code-in-arm-assembly-registers-explained/.

[13] Azeria. Memory Instructions: Load And Store. https://azeria-labs.com/memory-instructions-load-and-store-part-4/.

[14] Arm Developer. B, BL, BX, and BLX. https://developer.arm.com/documentation/dui0802/b/A32-and-T32-Instructions/B--BL--BX--and-BLX.

[15] Hopper. http://hopperapp.com/.

[16] Lambert, T. Clipping Silver Sparrow's wings: Outing macOS malware before it takes flight. Red Canary. February 2021. https://redcanary.com/blog/clipping-silver-sparrows-wings/.

[17] Karim, T. OSX/Hydromac: New Mac adware, leaked from a flashcards app. Objective-See. June 2021. https://objective-see.com/blog/blog_0x65.html.