2
PARSING BINARIES

In the previous chapter, we enumerated running processes and extracted information that could help us heuristically detect malware. However, we didn't cover how to examine the actual binary that backed each process. This chapter describes how to programmatically parse and analyze universal and Mach-O, the native executable binary file format of macOS.

You'll learn how to extract information such as a binary's dependencies and symbols, as well as detect whether the binary contains anomalies, such as encrypted data or instructions. This information will improve your ability to classify a binary as malicious or benign.

Universal Binaries

The majority of Mach-O binaries are distributed in universal binaries. Called *fat binaries* in Apple parlance, these are containers for multiple architecture-specific (but generally logically equivalent) Mach-O binaries known as *slices*. At runtime, the macOS dynamic loader (*dyld*) will load and then execute whichever embedded Mach-O binary best matches the host's native architecture (for example, Intel or ARM). Because these embedded binaries hold the information you're looking to extract, such as dependencies, you must first understand how to programmatically parse the universal binary.

Inspecting

Apple's `file` utility can inspect universal binaries. For example, the CloudMensis malware is distributed as a universal binary named *Window Server* containing two Mach-O binaries: one compiled for Intel x86_64 and one for Apple Silicon ARM64 systems. Let's execute `file` against CloudMensis. As you can see, the tool identifies it as a universal binary and shows its two embedded Mach-Os:

```
% file CloudMensis/WindowServer
CloudMensis/WindowServer: Mach-O universal binary with 2 architectures:
[x86_64:Mach-O 64-bit executable x86_64] [arm64:Mach-O 64-bit executable arm64]

CloudMensis/WindowServer (for architecture x86_64): Mach-O 64-bit executable x86_64
CloudMensis/WindowServer (for architecture arm64):  Mach-O 64-bit executable arm64
```

To programmatically access these embedded binaries, we have to parse the universal binary's header, which contains the offset of each Mach-O. Luckily, parsing the header is straightforward. Universal binaries start with a `fat_header` structure. We can find relevant universal structures and constants in Apple's SDK *mach-o/fat.h* header file:

```
struct fat_header {
    uint32_t    magic;        /* FAT_MAGIC or FAT_MAGIC_64 */
    uint32_t    nfat_arch;    /* number of structs that follow */
};
```

Apple's comments in this header file indicate that `magic`, the first member of the `fat_header` structure (an unsigned 32-bit integer), will contain the constant `FAT_MAGIC` or `FAT_MAGIC_64`. The use of `FAT_MAGIC_64` means the next structures are of the type `fat_arch_64`, used when the following slice or offset to it is greater than 4GB.[1] Comments in Apple's *fat.h* header files note that support for this extended format is a work in progress, and universal binaries are rarely, if ever, so massive, so we'll focus on the traditional `fat_arch` structure in this chapter.

Not mentioned in the `fat_header` structure's comments is the fact that the values in the structure are assumed to be big-endian, a vestige of the OSX PPC days. Therefore, on little-endian systems such as Intel and Apple Silicon, when you read a universal binary into memory, values such as the 4 bytes for `magic` will appear in reverse-byte order.

Apple accounts for this fact by providing the "swapped" magic constant `FAT_CIGAM`. (Yes, `CIGAM` is just magic backward.) The hexadecimal value of this constant is `0xbebafeca`.[2] We can see this value by using `xxd` to dump the bytes at the start of the CloudMensis universal binary. On a little-endian host, we make use of the `-e` flag to display the hexadecimal values in little-endian:

```
% xxd -e -c 4 -g 0 CloudMensis/WindowServer
00000000: bebafeca ...
...
```

The output, when interpreted as a 4-byte value, will have the host's endianness applied, which explains why we see the swapped universal magic value `FAT_CIGAM` (`0xbebafeca`).

Following the `magic` field in the `fat_header` structure, we find the `nfat_arch` field, which specifies the number of `fat_arch` structures. We'll find one `fat_arch` structure for each architecture-specific Mach-O binary embedded in the universal binary. As illustrated in Figure 2-1, these structures immediately follow the fat header.

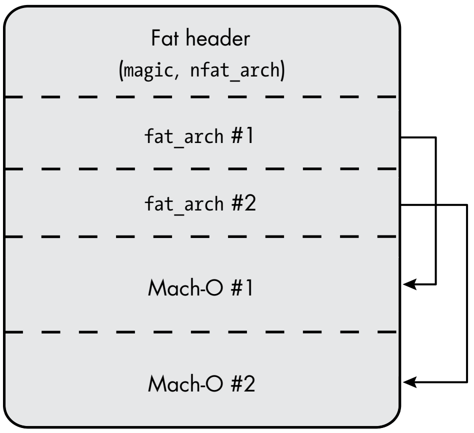


Figure 2-1: The layout of a universal binary

Because `file` showed that CloudMensis contained two embedded Mach-Os, we'd expect to see `nfat_arch` set to `2`. We confirm that this is the case by using `xxd` once again. This time, though, we skip the `-e` flag so as to keep the values in big endian:

```
% xxd -c 4 -g 0 CloudMensis/WindowServer
...
00000004: 00000002 ...
```

You can find the `fat_arch` structure definition in the *fat.h* header file:

```
struct fat_arch {
    cpu_type_t      cputype;       /* cpu specifier (int) */
    cpu_subtype_t   cpusubtype;    /* machine specifier (int) */
    uint32_t        offset;        /* file offset to this object file */
    uint32_t    size;     /* size of this object file */
    uint32_t    align;    /* alignment as a power of 2 */
};
```

The first two members of the `fat_arch` structure specify the CPU type and subtype of the Mach-O binary, while the next two specify the offset and size of this slice.

Parsing

Let's programmatically parse a universal binary and locate each embedded Mach-O binary. We'll show two methods of doing so: using the older `NX*` APIs compatible with older versions of macOS and the newer `Macho*` APIs available on macOS 13 and newer.

NOTE *You can find the code mentioned in this chapter in the* parseBinary *project in the book's GitHub repository at* https://github.com/Objective-see/TAOMM.

NX* APIs

We'll begin by checking whether the file is indeed a universal binary. Then we'll iterate over all `fat_arch` structures, printing out their values, and leverage the `NXFindBestFatArch` API to find the embedded binary most compatible with the host's architecture. The system will load and execute this binary when the universal binary is launched, so it's the one we'll focus on in our analysis.

Your own code may instead want to examine each embedded Mach-O binary, especially as nothing stops a developer from making these binaries completely different. Although you'll rarely find this to be the case, the 2023 3CX supply chain attack provides one notable exception. To trojanize the 3CX application, attackers subverted a legitimate universal binary that contained both Intel and ARM binaries, adding malicious code to the former and leaving the ARM binary untouched.

Let's start by loading a file and performing some initial checks (Listing 2-1).

```
#import <mach-o/fat.h>
#import <mach-o/arch.h>
#import <mach-o/swap.h>
#import <mach-o/loader.h>

int main(int argc, const char* argv[]) {

    NSData* data = [NSData dataWithContentsOfFile:[NSString stringWithUTF8String:argv[1]]]; ❶
    struct fat_header* fatHeader = (struct fat_header*)data.bytes; ❷

    if( (FAT_MAGIC == fatHeader->magic) || ❸
        (FAT_CIGAM == fatHeader->magic) ) {
        printf("\nBinary is universal (fat)\n");
        struct fat_arch* bestArch = parseFat(argv[1], fatHeader);
        ...
    }
    ...
}
```

Listing 2-1: Loading, validating, and finding the "best" slice of a universal binary

After reading the contents of the file into memory ❶ and typecasting the initial bytes to a `struct fat_header *` ❷, the code checks that it is indeed a universal binary ❸. Note that it checks both the big-endian (`FAT_MAGIC`) and little-endian (`FAT_CIGAM`) versions of the magic value.

To keep things simple, this code doesn't support the large fat file format. Moreover, for production code, you should perform other sanity checks, such as ensuring that the file was successfully loaded and that it's bigger than the size of a `fat_header` structure.

The parsing logic lives in a helper function named `parseFat`, which you can see invoked in Listing 2-1. After printing out the fat header, this function will iterate over each `fat_arch` structure and return the most compatible Mach-O slice.

First, though, we must deal with any differences in endianness. The values in the `fat_header` and `fat_arch` structures are always in big-endian order, so on little-endian systems such as Intel and Apple Silicon, we must swap them. To do so, we first invoke the `NXGetLocalArchInfo` API to determine the host's underlying byte order (Listing 2-2). We'll use the value returned, a pointer to an `NXArchInfo` structure, to swap the endianness (as well as later, to determine the most compatible Mach-O).

```
struct fat_arch* parseFat(const char* file, NSData* data) {
    const NXArchInfo* localArch = NXGetLocalArchInfo();
```

Listing 2-2: Determining the local machine's architecture

You might notice that the `NXGetLocalArchInfo` and `swap_*` APIs are marked as deprecated, although they're still available and fully functional at the time of publication. You can use replacement `macho_*` APIs, found in *mach-o/utils.h*, on macOS 13 and newer, and you'll learn about this in the next section. However, until macOS 15, one of these new APIs was broken, so you may still want to stick to the older APIs.

Next, we perform the swap with the `swap_fat_header` and `swap_fat_arch` functions (Listing 2-3).

```
struct fat_header* header = (struct fat_header*)data.bytes;

if(FAT_CIGAM == header->magic) { ❶
    swap_fat_header(header, localArch->byteorder); ❷
    swap_fat_arch((struct fat_arch*)((unsigned char*)header + sizeof(struct fat_header)),
    header->nfat_arch, localArch->byteorder); ❸
}
```

```
printf("Fat header\n");
printf("fat_magic %#x\n", header->magic);
printf("nfat_arch %d\n",  header->nfat_arch);
```

Listing 2-3: Swapping the fat header and fat architecture structures to match the host's byte ordering

The code first checks whether a swap is needed ❶. Recall that if the magic constant of the fat header is `FAT_CIGAM`, the code is executing on a little-endian host, so we should perform a swap. By invoking the helper APIs `swap_fat_header` ❷ and `swap_fat_arch` ❸, the code converts the header and all `fat_arch` values to match the host's byte ordering, as returned by `NXGetLocalArchInfo`. The latter API takes the number of `fat_arch` structures to swap, which the code provides via the `nfat_arch` field of the now-swapped fat header.

Once the header and all `fat_arch` structures conform to the host's byte ordering, the code can print out details of each embedded Mach-O binary that the `fat_arch` structures describe (Listing 2-4).

```
struct fat_arch* arch = (struct fat_arch*)((unsigned char*)header + sizeof(struct fat_header));

for(uint32_t i = 0; i < header->nfat_arch; i++) { ❶
    printf("architecture %d\n", i);
    printFatArch(&arch[i]);
}

void printFatArch(struct fat_arch* arch) { ❷
    int32_t cpusubtype = 0;
    cpusubtype = arch->cpusubtype & ~CPU_SUBTYPE_MASK; ❸

    printf(" cputype %u (%#x)\n", arch->cputype, arch->cputype);
    printf(" cpusubtype %u (%#x)\n", cpusubtype, cpusubtype);
    printf(" capabilities 0x%#x\n", (arch->cpusubtype & CPU_SUBTYPE_MASK) >> 24);
    printf(" offset %u (%#x)\n", arch->offset, arch->offset);
    printf(" size %u (%#x)\n", arch->size, arch->size);
    printf(" align 2^%u (%d)\n", arch->align, (int)pow(2, arch->align));
}
```

Listing 2-4: Printing out each `fat_arch` structure

The code starts by initializing a pointer to the first `fat_arch` structure, which comes immediately after the `fat_header`. Then it iterates over each, bounded by the `nfat_arch` member of the `fat_header` ❶. To print out values from each `fat_arch` structure, the code invokes a helper function we've named `printFatArch` ❷, which first separates the CPU subtype and its capabilities, as both are found in the `cpusubtype` member. Apple provides the `CPU_SUBTYPE_MASK` constant to extract just the bits that describe the subtype ❸.

Let's run this code against CloudMensis. It outputs the following:

```
% ./parseBinary CloudMensis/WindowServer
Binary is universal (fat)
Fat header
fat_magic 0xcafebabe
```

```
nfat_arch 2
architecture 0
    cputype 16777223 (0x1000007)
    cpusubtype 3 (0x3)
    capabilities 0x0
    offset 16384 (0x4000)
    size 708560 (0xacfd0)
    align 2^14 (16384)
architecture 1
    cputype 16777228 (0x100000c)
    cpusubtype 0 (0)
    capabilities 0x0
    offset 737280 (0xb4000)
    size 688176 (0xa8030)
    align 2^14 (16384)
```

From the output, we can see the malware's two embedded Mach-O binaries:

- At offset 16384, a binary compatible with `CPU_TYPE_X86_64` (`0x1000007`) that is 708,560 bytes long
- At offset 737280, a binary compatible with `CPU_TYPE_ARM64` (`0x100000c`) that is 688,176 bytes long

To confirm the accuracy of this code, we can compare this output against the macOS `otool` command, whose `-f` flag parses and displays fat headers:

```
% otool -f CloudMensis/WindowServer
Fat headers
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
    cputype 16777223
    cpusubtype 3
    capabilities 0x0
    offset 16384
    size 708560
    align 2^14 (16384)
architecture 1
    cputype 16777228
    cpusubtype 0
    capabilities 0x0
    offset 737280
    size 688176
    align 2^14 (16384)
```

In the tool's output, we see the same information about the malware's two embedded binaries.

Next, let's add some code to determine which of the embedded Mach-O binaries matches the host's native architecture. Recall that we already invoked the `NXGetLocalArchInfo` API to retrieve the host architecture. Moreover, we also showed how to compute the offset to the first `fat_arch` structure, which immediately follows the fat header. To find the natively compatible Mach-O, we can now invoke the `NXFindBestFatArch` API (Listing 2-5).

```
bestArchitecture = NXFindBestFatArch(localArch->cputype, localArch->
cpusubtype, arch, header->nfat_arch);
```

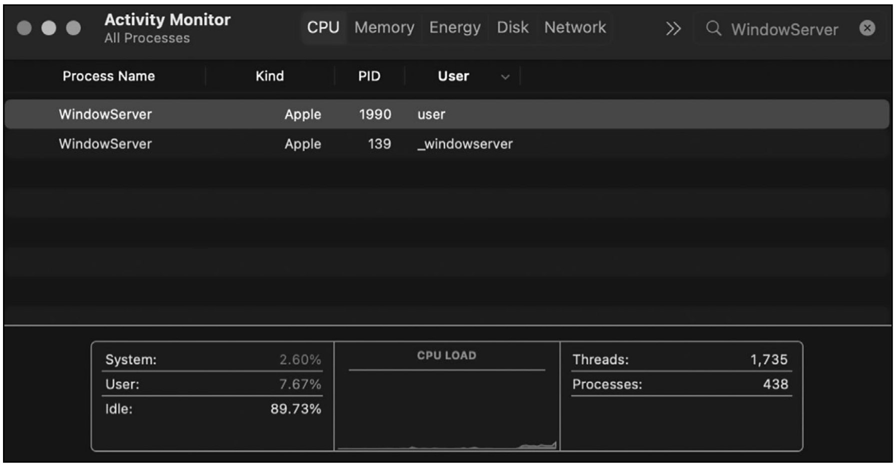
Listing 2-5: Determining a universal binary's best architecture

We pass the API the host's architecture, a pointer to the start of the `fat_arch` structures, and the number of these structures. The `NXFindBestFatArch` API will then determine the Mach-O binary from within the universal binary that is the most compatible with the host's native architecture. Recall the `parseFat` helper function returns this value and prints it out.

If we add this code to the binary parser and then run it again against CloudMensis, it outputs the following:

```
% ./parseBinary CloudMensis/WindowServer
...
best architecture match
    cputype 16777228 (0x100000c)
    cpusubtype 0 (0)
    capabilities 0x0
    offset 737280 (0xb4000)
    size 688176 (0xa8030)
    align 2^14 (16384)
```

On an Apple Silicon (ARM64) system, the code has correctly determined that the second embedded Mach-O binary, with a CPU type of `16777228/0x100000c` (`CPU_TYPE_ARM64`), is the most compatible Mach-O in the universal CloudMensis binary. When launching this universal binary, we can use the Kind column in Activity Monitor to confirm that macOS indeed selected and ran the Apple Silicon Mach-O (Figure 2-2).



Process Name	Kind	PID	User
WindowServer	Apple	1990	user
WindowServer	Apple	139	_windowserver

System:	2.60%
User:	7.67%
Idle:	89.73%

Threads:	1,735
Processes:	438

Figure 2-2: The CloudMensis binary WindowServer *running as a native Apple Silicon binary*

Another way to confirm that CloudMensis runs as a native Apple Silicon binary is to use the *enumerateProcesses* project presented in Chapter 1. Recall that it extracts the architecture of each running process:

```
% ./enumerateProcesses
...
(1990):/Library/WebServer/share/httpd/manual/WindowServer
...
architecture: Apple Silicon
```

We receive the same result.

Macho* APIs

In macOS 13, Apple introduced the macho_* APIs. Found in *mach-o/utils.h*, these APIs offer a simplified way to iterate over Mach-O binaries in a universal binary and select the most compatible one. The deprecated NX* APIs still work for this purpose, but if you're developing tools on macOS 13 or later, it's wise to instead use the newer functions.

The macho_for_each_slice API lets us extract a universal binary's Mach-Os without having to manually parse the universal header or deal with the nuances of byte orderings. We invoke this function with a path to a file and callback block to run for each Mach-O slice. If invoked against a stand-alone Mach-O, the function will run its callback just once, and if the file isn't a well-formed universal binary or Mach-O, the function will gracefully fail, meaning we don't have to manually verify the file type ourselves. The *mach-o/utils.h* header file includes the possible return values and their meanings:

```
ENOENT - path does not exist
EACCES - path exists but caller does not have permission to access it
EFTYPE - path exists but it is not a Mach-o or fat file
EBADMACHO - path is a Mach-o file, but it is malformed
```

The callback block invoked for each embedded Mach-O has the following type:

```
void (^ _Nullable callback)(const struct mach_header* _Nonnull slice,
uint64_t offset, size_t size, bool* _Nonnull stop)
```

This type might look a little confusing at first, but if we focus solely on the parameters, we see that the callback will be invoked with a variety of information about the slice, including a pointer to a mach_header structure, the slice's offset, and its size.

The code in Listing 2-6, part of the parseFat helper function, invokes macho_for_each_slice to print out information about each embedded Mach-O. It also includes some basic error handling, which we can use to filter out files that are neither universal nor Mach-Os.

```
struct fat_arch* parseFat(const char* file, struct fat_header* header) {
    ...
    if(@available(macOS 13.0, *)) {
        __block int count = 0;

        int result = macho_for_each_slice(file,
        ^(const struct mach_header* slice, uint64_t offset, size_t size, bool* stop) { ❶
            printf("architecture %d\n", count++); ❷
            printf("offset %llu (%#llx)\n", offset, offset);
            printf("size %zu (%#zx)\n", size, size);
            printf("name %s\n\n", macho_arch_name_for_mach_header(slice)); ❸
        });
        if(0 != result) {
            printf("ERROR: macho_for_each_slice failed\n");

            switch(result) { ❹
                case EFTYPE:
                    printf("EFTYPE: path exists but it is not a Mach-o or fat file\n\n");
                    break;

                case EBADMACHO:
                    printf("EBADMACHO: path is a Mach-o file, but it is malformed\n\n");
                    break;

                ...
            }
        }
    }
    ...
}
```

Listing 2-6: Iterating over all embedded Mach-Os

This code invokes the `macho_for_each_slice` function ❶. In the callback block, we print out a counter variable followed by the slice's offset and size ❷. We also make use of the `macho_arch_name_for_mach_header` function to print out the name of each slice's architecture ❸.

If the user-specified file isn't a well-formed universal or Mach-O binary, the function will fail. The code handles this, printing out a generic error message, as well as additional information for common errors ❹.

If we add this code to the *parseBinary* project and then run it against the CloudMensis universal binary, it should print out the same offset and size values for the malware's two embedded Mach-Os as the code that leveraged the `NX*` APIs:

```
% ./parseBinary CloudMensis/WindowServer
...
architecture 0
    offset 16384 (0x4000)
    size 708560 (0xacfd0)
    name x86_64
```

```
architecture 1
    offset 737280 (0xb4000)
    size 688176 (0xa8030)
    name arm64
```

Now, what about finding the most compatible slice, or the one that the host would load and run if the universal binary were executed? The `macho _best_slice` function is designed to return exactly that. It takes a path to a file to inspect and a callback block to invoke with the best slice. Add the function in Listing 2-7 to the previous code.

```
result = macho_best_slice(argv[1],
^(const struct mach_header* _Nonnull slice, uint64_t offset, size_t sliceSize) {
    printf("best architecture\n");
    printf("offset %llu (%#llx)\n", offset, offset);
    printf("size %zu (%#zx)\n", sliceSize, sliceSize);
    printf("name %s\n\n", macho_arch_name_for_mach_header(slice));
});
if(0 != result) {
    printf("ERROR: macho_best_slice failed with %d\n", result);
}
```

Listing 2-7: Invoking `macho_best_slice` to find the best slice

If we run this against CloudMensis (on a version of macOS prior to 15), however, it fails with the value `86`:

```
% ./parseBinary CloudMensis/WindowServer
...
ERROR: macho_best_slice failed with 86
```

According to the *mach-o/utils.h* header file, this error value maps to `EBADARCH`, which means none of the slices can load. This is odd, considering that the `NXFindBestFatArch` function identified the embedded ARM64 Mach-O binary as compatible with my Apple Silicon analysis machine. Moreover, this ARM64 Mach-O definitely runs, as you saw in Figure 2-2. It turns out, as is often the case with new APIs from Apple, that the `macho_best_slice` function was broken until macOS 15. On older versions of macOS, for any third-party universal binary on Apple Silicon systems, the function returns `EBADARCH`.

Reverse engineering, as well as studying the code of *dyld*,[3] revealed the cause of the error: instead of passing a list of compatible CPU types (such as `arm64` or `x86_64`) to the slice selection function, the code incorrectly passed in only the CPU type for which the operating system was compiled. On Apple Silicon, this CPU type is `arm64e` (`CPU_SUBTYPE_ARM64E`), used exclusively by Apple. This explains why the selection logic never chose slices in third-party universal binaries, which are compiled as `arm64` or `x86_64` (but never `arm64e`), and instead returned the `EBADARCH` error.

You can read more about the bug in my write-up "Apple Gets an 'F' for Slicing Apples."[4] My analysis proposed a simple fix: instead of invoking the `GradedArchs::forCurrentOS` method, Apple should have invoked `GradedArchs::launchCurrentOS` to obtain the correct list of compatible CPU types. The good news is that Apple eventually took this recommendation, meaning that `macho_best_slice` on macOS 15 and above works as expected.

Now that you know how to parse universal binaries, let's turn our attention to the Mach-O binaries embedded within them.[5]

Mach-O Headers

Mach-O binaries contain the information we're after, such as dependencies and symbols. To programmatically extract these, we must parse the Mach-O's header. In a universal binary, we can locate this header by analyzing the fat header and architecture structures, as you saw in the previous section. In a single-architecture, stand-alone Mach-O, finding the header is trivial, as it's located at the start of the file.

Listing 2-8 follows the code that identifies the best Mach-O within a universal binary. It confirms that the slice is indeed a Mach-O, then handles cases in which a file is a stand-alone Mach-O.

```
NSData* data = [NSData dataWithContentsOfFile:[NSString stringWithUTF8String:argv[1]]];

struct mach_header_64* machoHeader = (struct mach_header_64*)data.bytes; ❶

if( (FAT_MAGIC == fatHeader->magic) ||
    (FAT_CIGAM == fatHeader->magic) ) {
    // Removed the code that finds the best architecture, for brevity
    ...
    machoHeader = (struct mach_header_64*)(data.bytes + bestArch->offset); ❷
}

if( (MH_MAGIC_64 == machoHeader->magic) || ❸
    (MH_CIGAM_64 == machoHeader->magic) ) {
    printf("binary is Mach-O\n");
    // Add code here to parse the Mach-O.
}
```

Listing 2-8: Finding the relevant Mach-O header

After loading the file into memory, we typecast the bytes at the start of the file to a `mach_header_64` structure ❶. If the binary is universal, we find the `fat_arch` structure that describes the most compatible embedded Mach-O. Using this structure's `offset` member, we update the pointer to point to the embedded binary ❷.

Before we parse the binary, we must verify that the pointer really points to the start of the Mach-O. We take a simple verification approach: checking for the presence of a Mach-O magic value ❸. Because the binary's header and the host machine architecture could have different endianness, the code checks for both the `MH_MAGIC_64` and `MH_CIGAM_64` constants, defined in Apple's *mach-o/loader.h* header file:

```
#define MH_MAGIC_64 0xfeedfacf
#define MH_CIGAM_64 0xcffaedfe
```

For the sake of simplicity, the code skips recommended sanity and error checks. For example, production code should, at the very minimum, ensure that the size of the read-in bytes is greater than `sizeof(struct mach_header_64)` before dereferencing offsets in the header.

NOTE *Mach-O headers are of type* `mach_header` *or* `mach_header_64`*. Recent versions of macOS support 64-bit code only, so this section focuses on* `mach_header_64`*, defined in* mach-o/loader.h.

Now that we're sure we're looking at a Mach-O, we can parse it. Listing 2-9 defines a helper function named `parseMachO` for this purpose. It takes a pointer to the `mach_header_64` structure.

```
void parseMachO(struct mach_header_64* header) {
    if(MH_CIGAM_64 == machoHeader->magic) {
        swap_mach_header_64(machoHeader, ((NXArchInfo*)NXGetLocalArchInfo())->byteorder);
    }
    ...
}
```

Listing 2-9: Swapping the Mach-O header to match the host's byte ordering

Because the binary's header and the host machine could have a different endianness, the code first checks for the swapped Mach-O magic value. If you encounter it, swap the header via the `swap_mach_header_64` API. Note here that the code makes use of the macOS `NXGetLocalArchInfo` function, but if you're writing code for versions of macOS 13 or newer, you should use the more modern `macho*` APIs (again noting that the `macho_best_slice` function was broken until macOS 15).

To print out the Mach-O header, we write a helper function, `printMachOHeader` (Listing 2-10).

```
void printMachOHeader(struct mach_header_64* header) {
    int32_t cpusubtype = 0;
    cpusubtype = header->cpusubtype & ~CPU_SUBTYPE_MASK;

    printf("Mach-O header\n");
    printf(" magic %#x\n", header->magic);
    printf(" cputype %u (%#x)\n", header->cputype, header->cputype);
    printf(" cpusubtype %u (%#x)\n", cpusubtype, cpusubtype);
    printf(" capabilities %#x\n", (header->cpusubtype & CPU_SUBTYPE_MASK) >> 24);

    printf(" filetype %u (%#x)\n", header->filetype, header->filetype);

    printf(" ncmds %u\n", header->ncmds);
    printf(" sizeofcmds %u\n", header->sizeofcmds);

    printf(" flags %#x\n", header->flags);
}
```

Listing 2-10: Printing out a Mach-O header

You can find an overview of each header member in the comments of the mach_header_64 structure definition. For example, following the magic field are the two fields that describe the binary's compatible CPU type and subtype. The cpusubtype member also contains the binary's capabilities, and these can be extracted into their own field.

The file type indicates whether the binary is a stand-alone executable or a loadable library. The next fields describe the number and size of the binary's load commands, which we'll make extensive use of shortly. Finally, the flags member of the structure indicates additional optional features, such as whether the binary is compatible with address space layout randomization.

Let's run the Mach-O parsing code against CloudMensis. After searching the universal header, the tool finds the compatible Mach-O header and then prints it out:

```
% ./parseBinary CloudMensis/WindowServer
Mach-O header:
    magic 0xfeedfacf
    cputype 16777228 (0x100000c)
    cpusubtype 0 (0)
    capabilities 0
    filetype 2 (0x2)
    ncmds 28
    sizeofcmds 4192
    flags 0x200085
```

This output matches that of Apple's otool, whose -h flag instructs it to print out the Mach-O header:

```
% otool -h CloudMensis/WindowServer
...
CloudMensis/WindowServer (architecture arm64):
Mach header
 magic       cputype    cpusubtype   caps   filetype  ncmds  sizeofcmds  flags
 0xfeedfacf  16777228   0            0x00   2         28     4192        0x00200085
```

Running otool with the -v flag converts the returned numerical values into symbols:

```
% otool -hv CloudMensis/WindowServer
...
CloudMensis/WindowServer (architecture arm64):
Mach header
magic        cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64  ARM64   ALL        0x00 EXECUTE  28    4192       NOUNDEFS DYLDLINK
                                                               TWOLEVEL PIE
```

These values confirm that our tool works as expected.

Load Commands

Load commands are instructions to *dyld* that immediately follow the Mach-O header. A header field named ncmds specifies the number of load commands, and each command is a structure of type load_command containing the command type (cmd) and size (cmdsize), as you can see here:

```
struct load_command {
   uint32_t cmd;        /* type of load command */
   uint32_t cmdsize;    /* total size of command in bytes */
};
```

Some load commands describe the segments in the binary, such as the __TEXT segment that contains the binary's code, while others describe dependencies, the location of the symbol table, and more. As such, code that aims to extract information found within Mach-Os will generally start by parsing load commands.

Listing 2-11 defines a helper function named findLoadCommand for this purpose. It takes a pointer to a Mach-O header and the type of load command to find. After locating the start of the load commands, it iterates over each to create an array containing commands that match the specified type.

```
NSMutableArray* findLoadCommand(struct mach_header_64* header, uint32_t type) {
    NSMutableArray* commands = [NSMutableArray array];
    struct load_command* command = NULL;

    command = (struct load_command*)((unsigned char*)header + sizeof(struct mach_header_64)); ❶

    for(uint32_t i = 0; i < header->ncmds; i++) { ❷
        if(type == command->cmd) { ❸
            [commands addObject:[NSValue valueWithPointer:command]]; ❹
        }
        command = (struct load_command*)((unsigned char*)command + command->cmdsize); ❺
    }

    return commands;
}
```

Listing 2-11: Iterating over all load commands and collecting those that match a specified type

We start by calculating a pointer to the first load command, which immediately follows the Mach-O header ❶. Then we iterate over all load commands, which appear one after another ❷, and check the cmd member of each to see if it matches the specified type ❸. As we can't directly store pointers in an Objective-C array, we first create an NSValue object with the load command's address ❹. Finally, we advance to the next load command. Load commands can vary in size, so we use the current command's cmdsize field ❺ to find the next one.

With an understanding of load commands and a helper function that returns commands of interest, let's now consider a few examples of pertinent information we can extract, starting with dependencies.

Extracting Dependencies

One of the reasons to parse Mach-Os is to extract their *dependencies*: dynamic libraries that *dyld* will automatically load. Understanding the dependencies of a binary can provide insight into its likely capabilities or even uncover malicious dependencies. For example, CloudMensis links against the *DiskArbitration* framework, which provides APIs to interact with external disks. Using this framework's APIs, the malware monitors for the insertion of removable USB drives so it can exfiltrate external files.

When writing code, we can often achieve the same outcome in several ways. For example, in Chapter 1, we extracted all loaded libraries and frameworks from a running process by leveraging `vmmap`. In this chapter, we'll perform a similar task by manually parsing the Mach-O. This static approach will extract direct dependencies only, excluding recursion; that is to say, we won't extract the dependencies of dependencies. Moreover, libraries directly loaded by the binary at runtime are not dependencies per se and thus will not be extracted. While simple, this technique should help us understand the Mach-O's capabilities and doesn't require executing external binaries like `vmmap`. Also, the code will run against any Mach-O binary without requiring it to be currently executing.

Finding Dependency Paths

To extract a binary's dependencies, we can enumerate its `LC_LOAD_DYLIB` load commands, each of which contains a path to a library or framework on which the Mach-O depends. The `dylib_command` structure describes these load commands:

```
struct dylib_command {
    uint32_t      cmd;         /* LC_ID_DYLIB, LC_LOAD_{,WEAK_}DYLIB, LC_REEXPORT_DYLIB */
    uint32_t      cmdsize;     /* includes pathname string */
    struct dylib  dylib;       /* the library identification */
};
```

We'll extract these dependencies in a function named `extractDependencies` that accepts a pointer to a Mach-O header and returns an array containing the names of dependencies.

NOTE *To keep things simple, we won't take into account `LC_LOAD_WEAK_DYLIB` load commands, which describe optional dependencies.*

In Listing 2-12, the code starts by invoking the `findLoadCommand` helper function to find load commands whose type is `LC_LOAD_DYLIB`. It then iterates over each of these load commands to extract the dependency's path.

```
NSMutableArray* extractDependencies(struct mach_header_64* header) {
    ...
    NSMutableArray* commands = findLoadCommand(header, LC_LOAD_DYLIB);

    for(NSValue* command in commands) {
        // Add code here to extract each dependency.
    }
```

Listing 2-12: Finding all `LC_LOAD_DYLIB` load commands

Let's now extract the name of each dependency. To understand how we'll do so, take a look at the `dylib` structure that describes a dependency. This structure is the last member of the `dylib_command` structure used to describe `LC_LOAD_DYLIB` load commands:

```
struct dylib {
    union lc_str  name;                 /* library's path name */
    uint32_t timestamp;                 /* library's build time stamp */
    uint32_t current_version;           /* library's current version number */
    uint32_t compatibility_version;     /* library's compatibility vers number*/
};
```

Of interest to us is the structure's `name` field, whose type is `lc_str`. A comment in Apple's *loader.h* file explains that we must first extract the offset to the dependency path and then use it to compute the path's bytes and length (Listing 2-13).

```
NSMutableArray* dependencies = [NSMutableArray array];

for(NSValue* command in commands) {
    struct dylib_command* dependency = command.pointerValue; ❶

    uint32_t offset = dependency->dylib.name.offset; ❷
    char* bytes = (char*)dependency + offset;
    NSUInteger length = dependency->cmdsize-offset;

    NSString* path = [[NSString alloc] initWithBytes:bytes length:length encoding:NSUTF8
    StringEncoding]; ❸

    [dependencies addObject:path];
}
```

Listing 2-13: Extracting a dependency from an `LC_LOAD_DYLIB` load command

We previously stored the pointer to each matching load command as an `NSValue` object, so we must first extract these ❶. Then we extract the offset to the dependency path and use it to compute the path's bytes and length ❷. Now we can easily extract the path into a string object and save it into an array ❸. We return this array containing all dependencies once the enumeration is complete.

When we compile and run this code against CloudMensis, it outputs the following:

```
% ./parseBinary CloudMensis/WindowServer
...
Dependencies: (count: 12): (
    ...
    "/usr/lib/libobjc.A.dylib",
    "/usr/lib/libSystem.B.dylib",
    ...
    "/System/Library/Frameworks/DiskArbitration.framework/Versions/A/DiskArbitration",
    "/System/Library/Frameworks/SystemConfiguration.framework/Versions/A/SystemConfiguration"
)
```

Notice the inclusion of the *DiskArbitration* framework we mentioned earlier. Once again, we can use `otool`, this time with the `-L` flag, to confirm the accuracy of our code:

```
% otool -L CloudMensis/WindowServer
...
"/usr/lib/libobjc.A.dylib",
"/usr/lib/libSystem.B.dylib",
...
"/System/Library/Frameworks/DiskArbitration.framework/Versions/A/DiskArbitration",
"/System/Library/Frameworks/SystemConfiguration.framework/Versions/A/SystemConfiguration"
```

The dependencies extracted from CloudMensis via `otool` match those extracted by our code, so we can move on to analyzing them.

Analyzing Dependencies

The majority of CloudMensis's dependencies are system libraries and frameworks, such as *libobjc.A.dylib* and *libSystem.B.dylib*. Essentially all Mach-O binaries link against these, and from the point of view of malware detection, they're uninteresting. However, the *DiskArbitration* dependency is notable, as it provides the `DA*` APIs to interact with external disks. Here is a snippet of CloudMensis's decompiled binary code showing its interactions with the *DiskArbitration* APIs:

```
-(void)loop_usb {
    rax = DASessionCreate(**_kCFAllocatorDefault);
  ❶ DARegisterDiskAppearedCallback(rax, 0x0, OnDiskAppeared, 0x0);
    ...
}

int OnDiskAppeared() {
    ...
  ❷ r13 = DADiskCopyDescription(rdi);
    rax = CFDictionaryGetValue(r13, **_kDADiskDescriptionVolumeNameKey);
    r14 = [NSString stringWithFormat:@"/Volumes/%@", rax];
    ...

    rax = [functions alloc];
    r15 = [rax randPathWithPrefix:0x64 isZip:0x0];
    rax = [FileTreeXML alloc];
    [rax startFileTree:r14 dropPath:r15];
    ...
    [rax MoveToFileStore:r15 Copy:0x0];
    rax = [NSURL fileURLWithPath:r14];
    r14 = [NSMutableArray arrayWithObject:rax];

    rax = [functions alloc];
    [rax SearchAndMoveFS:r14 removable:0x1];
    ...
}
```

First, in a function named `loop_usb`, the malware invokes various `DiskArbitration` APIs to register a callback that the operating system will invoke automatically once a new disk appears ❶. When this `OnDiskAppeared` callback is invoked—for example, when an external USB drive is inserted—it calls other `DA*` APIs, such as `DADiskCopyDescription` ❷, to access information about the new disk. The remainder of the code in the `OnDiskAppeared` callback is responsible for generating a file listing, then copying files off the drive into a custom file store. These files eventually get exfiltrated to the attacker's remote command-and-control server.

Let's run the dependency code against another malware sample that leverages even more frameworks to achieve a wide range of offensive capabilities. Mokes is a cross-platform cyber-espionage implant that has infected macOS users in attacks leveraging browser zero-days.[6] Running the dependency extractor code against the malware's binary, named *storeuserd*, generates the following output:

```
% ./parseBinary Mokes/storeuserd
...
Dependencies: (count: 25): (
    "/System/Library/Frameworks/DiskArbitration.framework/Versions/A/DiskArbitration",
    "/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit",
    "/System/Library/Frameworks/ApplicationServices.framework/Versions/A/ApplicationServices",
    "/System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices",
    "/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation",
    "/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation",
    "/System/Library/Frameworks/Security.framework/Versions/A/Security",
    "/System/Library/Frameworks/SystemConfiguration.framework/Versions/A/SystemConfiguration",
    "/System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa",
    "/System/Library/Frameworks/Carbon.framework/Versions/A/Carbon",
    "/System/Library/Frameworks/AudioToolbox.framework/Versions/A/AudioToolbox",
    "/System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio",
    "/System/Library/Frameworks/QuartzCore.framework/Versions/A/QuartzCore",
    "/System/Library/Frameworks/AVFoundation.framework/Versions/A/AVFoundation",
    "/System/Library/Frameworks/CoreMedia.framework/Versions/A/CoreMedia",
    "/System/Library/Frameworks/AppKit.framework/Versions/C/AppKit",
    "/System/Library/Frameworks/AudioUnit.framework/Versions/A/AudioUnit",
    "/System/Library/Frameworks/CoreWLAN.framework/Versions/A/CoreWLAN",
    ...
)
```

Several of these dependencies shed light on the malware's capabilities and could guide future analysis. For example, the malware leverages the *AVFoundation* framework to record audio and video from the mic and webcam of an infected host. It also uses CoreWLAN to enumerate and monitor network interfaces and DiskArbitration to monitor external storage drives to find and exfiltrate files of interest.

Of course, dependencies alone can't prove that code is malicious. For example, a binary that links against the AVFoundation isn't necessarily spying on the user; it might be a legitimate videoconferencing app or simply be making use of the framework for benign multimedia-related tasks. However, taking a look at the following snippet of disassembly from Mokes confirms that it does indeed leverage AVFoundation APIs in a nefarious manner:

```
rax = AVFAudioInputSelectorControl::createCaptureDevice();
...
rax = [AVCaptureDeviceInput deviceInputWithDevice:rax error:&var_28];
...
QMetaObject::tr(..., "Could not connect the video recorder");
```

This excerpt shows the code interfacing with the webcam to spy on victims.

Another reason to extract dependencies from a Mach-O binary is to detect malicious subversions. ZuRu is one such example. Its malware authors surreptitiously trojanized popular applications such as iTerm by adding a malicious dependency to them, then distributed the applications via sponsored ads that would appear as the first result when users searched online for the applications.

The subversion was stealthy, as it left the original application's functionality wholly intact. However, extracting dependencies quickly reveals the malicious dependency. To demonstrate this, let's first extract the dependencies from a legitimate copy of iTerm2:

```
% ./parseBinary /Applications/iTerm.app/Contents/MacOS/iTerm2
...
Dependencies: (count: 33):
    "/usr/lib/libaprutil-1.0.dylib",
    "/usr/lib/libicucore.A.dylib",
    "/usr/lib/libc++.1.dylib",
    "@rpath/BetterFontPicker.framework/Versions/A/BetterFontPicker",
    "@rpath/SearchableComboListView.framework/Versions/A/SearchableComboListView",
    "/System/Library/Frameworks/OpenDirectory.framework/Versions/A/OpenDirectory",
    ...
    "/System/Library/Frameworks/QuartzCore.framework/Versions/A/QuartzCore",
    "/System/Library/Frameworks/WebKit.framework/Versions/A/WebKit",
    "/usr/lib/libsqlite3.dylib",
    "/usr/lib/libz.1.dylib"
)
```

Nothing unusual here. Now, if we extract the dependencies from a trojanized instance of iTerm, we uncover a new dependency, *libcrypto.2.dylib*, located in the application bundle. This dependency sticks out, not only because it doesn't exist in the legitimate application but also because it's the only dependency that uses the @executable_path variable:

```
% ./parseBinary ZuRu/iTerm.app/Contents/MacOS/iTerm2
...
Dependencies: (count: 34):
    "/usr/lib/libaprutil-1.0.dylib",
    "/usr/lib/libicucore.A.dylib",
    "/usr/lib/libc++.1.dylib",
    "@rpath/BetterFontPicker.framework/Versions/A/BetterFontPicker",
    "@rpath/SearchableComboListView.framework/Versions/A/SearchableComboListView",
    "/System/Library/Frameworks/OpenDirectory.framework/Versions/A/OpenDirectory",
    ...
    "/System/Library/Frameworks/QuartzCore.framework/Versions/A/QuartzCore",
    "/System/Library/Frameworks/WebKit.framework/Versions/A/WebKit",
    "/usr/lib/libsqlite3.dylib",
    "/usr/lib/libz.1.dylib",
    "@executable_path/../Frameworks/libcrypto.2.dylib"
)
```

There is nothing inherently malicious about the @executable_path variable; it simply tells the loader how to relatively resolve the library's path (meaning the library is likely embedded in the same bundle as the executable). Nevertheless, the addition of a new dependency that referenced a newly added library clearly warranted additional analysis, and such analysis revealed that the dependency contained all of the malware's malicious logic.[7]

Extracting Symbols

A binary's symbols contain the names of the binary's functions or methods and those of the APIs it imports. These function names can reveal the file's capabilities and even provide indicators that it is malicious. For example, let's extract the symbols from malware called DazzleSpy using the macOS nm tool:

```
% nm DazzleSpy/softwareupdate
...
"+[Exec doShellInCmd:]",
"-[ShellClassObject startPty]",
"-[MethodClass getIPAddress]",
"-[MouseClassObject PostMouseEvent::::]",
"-[KeychainClassObject getPasswordFromSecKeychainItemRef:]"
...
```

From the format of these symbols, we can tell that the malware was written in Objective-C. The Objective-C runtime requires method names to remain intact in the compiled binary, so understanding the binaries' capabilities is often relatively easy. For example, the symbols embedded in DazzleSpy reveal methods that appear to execute shell commands, survey the system, post mouse events, and steal passwords from the keychain.

It's worth noting, though, that nothing stops malware authors from using misleading method names, so you should never draw conclusions solely from extracted symbols. You might also encounter symbols that have been obfuscated (providing a pretty good indication that the binary has something to hide). Finally, the authors may have stripped a binary to remove symbols that aren't essential for program execution.

Later in the `nm` symbol output for DazzleSpy, we also find APIs that the malware imports from system libraries and frameworks:

```
_bind
_connect
_AVMediaTypeVideo
_AVCaptureSessionRuntimeErrorNotification
_NSFullUserName
_SecKeychainItemCopyContent
```

These include networking APIs such as `bind` and `connect` related to the malware's backdoor capabilities, `AVFoundation` imports related to its remote desktop capabilities, and APIs to survey a system and grab items from the victim's keychain.

How can we extract a Mach-O binary's symbols programmatically? As you'll see, this requires yet again parsing the binary's load commands. We'll focus specifically on the `LC_SYMTAB` load command, which contains information about a binary's symbols found in the symbol table (hence the load command's suffix `SYMTAB`). This load command consists of a `symtab_command` structure, defined in *loader.h*:

```
struct symtab_command {
    uint32_t       cmd;           /* LC_SYMTAB */
    uint32_t       cmdsize;       /* sizeof(struct symtab_command) */
    uint32_t       symoff;        /* symbol table offset */
    uint32_t       nsyms;         /* number of symbol table entries */
    uint32_t       stroff;        /* string table offset */
    uint32_t       strsize;       /* string table size in bytes */
};
```

The `symoff` member contains the offset of the symbol table, while `nsyms` contains the number of entries in this table. The symbol table consists of `nlist_64` structures, defined in *nlist.h*:

```
struct nlist_64 {
    union {
        uint32_t  n_strx;  /* index into the string table */
    } n_un;
    uint8_t n_type;        /* type flag, see below */
    uint8_t n_sect;        /* section number or NO_SECT */
    uint16_t n_desc;       /* see <mach-o/stab.h> */
    uint64_t n_value;      /* value of this symbol (or stab offset) */
};
```

Each nlist_64 structure in the symbol table contains an index to the string table, in the n_strx field. We can find the string table's offset in the symtab_command structure's stroff field. By adding the specified index from n_strx to this offset, we can retrieve the symbol as a NULL-terminated string. Thus, to extract a binary's symbols, we must perform the following steps:

1. Find the LC_SYMTAB load command that contains the symtab_command structure.
2. Use the symoff member of the symtab_command structure to find the offset of the symbol table.
3. Use the stroff member of the symtab_command structure to find the offset of the string table.
4. Iterate through all of the symbol table's nlist_64 structures to extract each symbol's index (n_strx) into the string table.
5. Apply this index to the string table to find the name of the symbol.

The function in Listing 2-14 implements these steps. Given a pointer to a Mach-O header, it saves all symbols into an array and returns it to the caller.

```
NSMutableArray* extractSymbols(struct mach_header_64* header) {
    NSMutableArray* symbols = [NSMutableArray array];

    NSMutableArray* commands = findLoadCommand(header, LC_SYMTAB);
    struct symtab_command* symTableCmd = ((NSValue*)commands.firstObject).pointerValue; ❶

    void* symbolTable = (((void*)header) + symTableCmd->symoff); ❷
    void* stringTable = (((void*)header) + symTableCmd->stroff); ❸
    struct nlist_64* nlist = (struct nlist_64*)symbolTable; ❹
    for(uint32_t j = 0; j < symTableCmd->nsyms; j++) { ❺
        char* symbol = (char*)stringTable + nlist->n_un.n_strx; ❻
        if(0 != symbol[0]) {
            [symbols addObject:[NSString stringWithUTF8String:symbol]];
        }
        nlist++;
    }
    return symbols;
}
```

Listing 2-14: Extracting a binary's symbols

Because this function is somewhat involved, we'll walk through it in detail. First, it finds the LC_SYMTAB load command by means of the findLoadCommand helper function ❶. It then uses the fields in the load command's symtab_command structure to compute the in-memory address of both the symbol table ❷ and the string table ❸. After initializing a pointer to the first nlist_64 structure, found at the start of the symbol table ❹, the code iterates over it and all subsequent nlist_64 structures ❺. For each of these structures, it adds the index to the string table to compute the address of the symbol's string representation ❻. If the symbol is not NULL, the code adds it to an array to return to the caller.

Let's compile and run this code against DazzleSpy. As we can see, the code is able to extract the malware's method names, as well as the API imports it invokes:

```
% ./parseBinary DazzleSpy/softwareupdate
...
Symbols (count: 3101): (

"-[ShellClassObject startPty]",
"-[ShellClassObject startTask]",

"-[MethodClass getDiskSize]",
"-[MethodClass getDiskFreeSize]",
"-[MethodClass getDiskSystemSize]",
"-[MethodClass getAllhardwareports]",
"-[MethodClass getIPAddress]",

"-[MouseClassObject PostMouseEvent::::]",
"-[MouseClassObject postScrollEvent:]",

"-[KeychainClassObject getPass:cmdTo:]",
"-[KeychainClassObject getPasswordFromSecKeychainItemRef:]",

"_bind",
"_connect",
...
"_AVMediaTypeVideo",
"_AVCaptureSessionRuntimeErrorNotification",
)
```

The ability to extract symbols from any Mach-O binary will improve our heuristic malware detection. Next, we'll programmatically detect anomalous characteristics that often indicate a binary is up to something nefarious.

NOTE *Newer binaries may contain a* `LC_DYLD_CHAINED_FIXUPS` *load command that optimizes how symbols and imports are handled on recent versions of macOS. In this case, a different approach is needed to extract embedded symbols. See the* `extractChainedSymbols` *function in the* parseBinary *project for more details and a programmatic implementation of such extraction.*

Detecting Packed Binaries

An *executable packer* is a tool that compresses binary code to shrink its size for distribution. The packer inserts a small unpacker stub at the binary's entry point, and this stub executes automatically when the packed program is run, restoring the original code in memory.

Malware authors are quite fond of packers, as compressed code is more difficult to analyze. Moreover, certain packers encrypt or further obfuscate the binary in an attempt to thwart signature-based detections and complicate analysis. Legitimate software is rarely packed on macOS, so the ability to detect obfuscation can be a powerful heuristic for flagging binaries that warrant closer inspection.

I'll wrap up this chapter by showing how to detect packed and encrypted Mach-O binaries by looking for a lack of dependencies and symbols, anomalous section and segment names, and high entropy.

Dependencies and Symbols

One simple, albeit somewhat naive, approach to packer detection is enumerating a binary's dependencies and symbols—or, rather, lack thereof. Nonpacked binaries will always have dependencies on various system frameworks and libraries such as *libSystem.B.dylib*, as well as imports from these dependencies. Packed binaries, on the other hand, may lack even a single dependency or symbol, as the unpacker stub will dynamically resolve and load any required libraries.

A binary with no dependencies or symbols is, at the very least, anomalous, and our tool should flag it for analysis. For example, running the dependency and symbol extraction code against the oRAT malware finds no dependencies or symbols:

```
% ./parseBinary oRat/darwinx64
...
Dependencies: (count: 0): ( )
Symbols: (count: 0): ( )
```

Apple's `otool` and `nm` confirm this absence as well:

```
% otool -L oRat/darwinx64
oRat/darwinx64:

% nm oRat/darwinx64
oRat/darwinx64: no symbols
```

It turns out oRAT is packed via UPX, a cross-platform packer that Mac malware authors favor. Examples of other macOS malware packed with UPX include IPStorm, ZuRu, and Coldroot.

Section and Segment Names

Binaries packed with UPX may contain UPX-specific section or segment names, such as `__XHDR`, `UPX_DATA`, or `upxTEXT`. If we find these names when parsing a Mach-O binary's segments, we can conclude that the binary was packed. Other packers, such as MPress, add their own segment names, such as `__MPRESS__`.

The following code snippet, from UPX's *p_mach.cpp* file,[8] shows references to nonstandard segment names:

```
if (!strcmp("__XHDR", segptr->segname)) {
    // PackHeader precedes __LINKEDIT
    style = 391;  // UPX 3.91
}
```

```
if (!strcmp("__TEXT", segptr->segname)) {
    ptrTEXT = segptr;
    style = 391;  // UPX 3.91
}
if (!strcmp("UPX_DATA", segptr->segname)) {
    // PackHeader follows loader at __LINKEDIT
    style = 392;  // UPX 3.92
}
```

To retrieve a binary's section and segment names, we can iterate through its load commands, looking for those of type `LC_SEGMENT_64`. These load commands consist of `segment_command_64` structures that contain a member named `segname` with the name of the segment. Here is the `segment_command_64` structure:

```
struct segment_command_64 { /* for 64-bit architectures */
    uint32_t        cmd;            /* LC_SEGMENT_64 */
    uint32_t        cmdsize;        /* includes sizeof section_64 structs */
    char            segname[16];    /* segment name */
    ...
    uint32_t        nsects;         /* number of sections in segment */
    uint32_t        flags;          /* flags */
};
```

Any sections within the segment should immediately follow the `segment_command_64` structure, whose `nsects` member specifies the number of sections. The `section_64` structure, shown here, describes sections:

```
struct section_64 { /* for 64-bit architectures */
    char            sectname[16];   /* name of this section */
    char            segname[16];    /* segment this section goes in */
    ...
};
```

Since the segment name can be extracted from the `segment_command_64` structure, here we're solely interested in the section name, `sectname`. To detect packers such as UPX, our code can iterate through each segment and its sections, comparing the names with those of common packers. First, though, we need a function that accepts a Mach-O header, then extracts the binary's segments and sections. The `extractSegmentsAndSections` function partially shown in Listing 2-15 does exactly this.

```
NSMutableArray* extractSegmentsAndSections(struct mach_header_64* header) {

    NSMutableArray* names = [NSMutableArray array];
    NSCharacterSet* nullCharacterSet = [NSCharacterSet
    characterSetWithCharactersInString:@"\0"];

    NSMutableArray* commands = findLoadCommand(header, LC_SEGMENT_64);
    for(NSValue* command in commands) {
        // Add code here to iterate over each segment and its sections.
    }
    return names;
}
```

Listing 2-15: Retrieving a list of LC_SEGMENT_64 load commands

This code declares a few variables and then invokes the now-familiar findLoadCommand helper function with a value of LC_SEGMENT_64. Now that we have a list of the load commands describing each segment in the binary, we can iterate over each, saving their names and the names of all their sections (Listing 2-16).

```
NSMutableArray* extractSegmentsAndSections(struct mach_header_64* header) {
    NSMutableArray* names = [NSMutableArray array];
    ...

    for(NSValue* command in commands) {
        struct segment_command_64* segment = command.pointerValue; ❶

        NSString* name = [[NSString alloc] initWithBytes:segment->segname
        length:sizeof(segment->segname) encoding:NSASCIIStringEncoding]; ❷

        name = [name stringByTrimmingCharactersInSet:nullCharacterSet];
        [names addObject:name];

        struct section_64* section = (struct section_64*)((unsigned char*)segment +
        sizeof(struct segment_command_64)); ❸

        for(uint32_t i = 0; i < segment->nsects; i++) { ❹
            name = [[NSString alloc] initWithBytes:section->sectname
            length:sizeof(section->sectname) encoding:NSASCIIStringEncoding]; ❺

            name = [name stringByTrimmingCharactersInSet:nullCharacterSet];
            [names addObject:name];

            section++;
        }
    }
    return names;
}
```

Listing 2-16: Iterating over each segment and its sections to extract their names

After extracting the pointer to each LC_SEGMENT_64 and saving it into a struct segment_command_64* ❶, the code extracts the name of the segment from the segname member of the segment_command_64 structure, stored in a rather unwieldy (and not necessarily NULL-terminated) char array. The code converts it into a string object, trims any NULLs, and then saves it into an array to return to the caller ❷.

Next, we iterate over the section_64 structures found in the LC_SEGMENT_64 command. One structure exists for each section in the segment. Because they begin immediately after the segment_command_64 structure, we initialize a pointer to the first section_64 structure, adding the start of the segment_command_64 structure to the size of this structure ❸. Now we can iterate over each section structure, bounded by the `nsects` member of the segment structure ❹. As with each segment name, we extract, convert, trim, and save the section names ❺.

Once we've extracted all segment and section names, we pass this list to a simple helper function named `isPacked`. Shown in Listing 2-17, it checks whether any names match those of well-known packers, such as UPX and MPress.

```
NSMutableSet* isPacked(NSMutableArray* segsAndSects) {
    NSSet* packers = [NSSet setWithObjects:@"__XHDR", @"upxTEXT", @"__MPRESS__", nil]; ❶

    NSMutableSet* packedNames = [NSMutableSet setWithArray:segsAndSects]; ❷
    [packedNames intersectSet:packers]; ❸

    return packedNames;
}
```

Listing 2-17: Checking for segment and section names matching those of known packers

First, we initialize a set with a few well-known packer-related segment and section names ❶. Then we convert the list of segments and sections into a mutable set ❷, as mutable set objects support the `intersectSet:` method, which will remove any items in the first set that aren't in the second. Once we've called this method ❸, the only names left in the set of segment and section names will match the packer-related ones.

After adding this code to the *parseBinary* project, we can run it against the macOS variant of the IPStorm malware:

```
% ./parseBinary IPStorm/IPStorm
binary is Mach-O
...
segments and sections: (
    "__PAGEZERO",
    "__TEXT",
    "upxTEXT",
    "__LINKEDIT"
)

binary appears to be packed
packer-related section or segment {( upxTEXT )} detected
```

Because the IPStorm binary contains a section named `upxTEXT` indicative of UPX, our code correctly ascertains that the binary is packed.

This name-based approach to packer detection has a low false-positive detection rate. However, it won't detect custom packers or even modified versions of known packers. For example, if an attacker modifies UPX to remove custom section names (which, as UPX is open source, is easy to do), we'll have a false negative, and the packed binary won't be detected.

We find an example of this behavior in the malware known as OceanLotus. In variant *H*, its authors packed the binary, *flashlightd*, with a customized version of UPX. Our current packer detector fails to determine that the malware is packed:

```
% ./parseBinary OceanLotus.H/flashlightd
binary is Mach-O
...
segments and sections: (
    "__PAGEZERO",
    "__TEXT",
    "__cfstring",
    "__LINKEDIT"
)

binary does not appear to be packed
no packer-related sections or segments detected
```

However, if we manually examine the malware, it becomes fairly obvious that the binary is packed. In a disassembler, large chunks of the binary appear obfuscated. We can also see that the binary contains no symbols or dependencies:

```
% ./parseBinary OceanLotus.H/flashlightd
binary is Mach-O
...
Dependencies: (count: 0): ()
Symbols: (count: 0): ()
```

Clearly, our packer detection approach needs some improvement. You'll see how to detect packed binaries via their entropy next.

Entropy Calculations

When a binary is packed, the amount of randomness in it greatly increases. This is largely due to the fact that packers either compress or encrypt the binary's original instructions. If we can calculate a binary's quantity of unique bytes and classify it as anomalously high, we can fairly accurately conclude the binary is packed.

Let's parse a Mach-O binary and calculate the entropy of its executable segments. The code in Listing 2-18 builds on the segment parsing code in the `isPackedByEntropy` function. After enumerating all `LC_SEGMENT_64` load commands, the function invokes a helper function named `calcEntropy` on each to calculate the entropy of the segment's data.

```
float calcEntropy(unsigned char* data, NSUInteger length) {
    float pX = 0.0f;
    float entropy = 0.0f;
    unsigned int occurrences[256] = {0};

    for(NSUInteger i = 0; i < length; i++) {
      ❶ occurrences[0xFF & (int)data[i]]++;
    }

    for(NSUInteger i = 0; i < sizeof(occurrences)/sizeof(occurrences[0]); i++) {
      ❷ if(0 == occurrences[i]) {
            continue;
        }

      ❸ pX = occurrences[i]/(float)length;
                    entropy -= pX*log2(pX);
    }
    return entropy;
}
```

Listing 2-18: Computing the Shannon entropy

The function first computes the number of occurrences of each byte value, from `0` to `0xFF` ❶. After skipping values that don't occur ❷, it performs a standard formula ❸ to compute the Shannon entropy.[9] The function should return a value between 0.0 and 8.0, ranging from no entropy (meaning all the values are the same) to the highest level of entropy.[10]

The code uses the entropy to determine whether the binary is likely packed (Listing 2-19). It's inspired by the popular Windows-centric AnalyzePE and pefile Python libraries.[11]

```
BOOL isPackedByEntropy(struct mach_header_64* header, NSUInteger size) {
    ...
    BOOL isPacked = NO;
    float compressedData = 0.0f;

    NSMutableArray* commands = findLoadCommand(header, LC_SEGMENT_64);
    for(NSValue* command in commands) {
        ...
        struct segment_command_64* segment = command.pointerValue;

        float segmentEntropy = calcEntropy(((unsigned char*)header +
        segment->fileoff), segment->filesize);

      ❶ if(segmentEntropy > 7.0f) {
            compressedData += segment->filesize;
        }
    }

  ❷ if((compressedData/size) > .2) {
        isPacked = YES;
    }
    ...
    return isPacked;
}
```

Listing 2-19: Packer detection via entropy analysis

Testing has shown that if the entropy of an average-size segment is above 7.0, we can confidently conclude that the segment contains compressed data, meaning it's either packed or encrypted. In this case, we append the segment's size to a variable to keep track of the total amount of compressed data ❶.

Once we've computed the entropy of each segment, we check how much of the binary's total data is packed by dividing the amount of compressed data by the size of the Mach-O. Research has shown that Mach-O binaries with a ratio of packed data to overall length greater than 20 percent are likely packed (though the ratio is usually much higher) ❷.

Let's test this code against the packed IPStorm sample:

```
% ./parseBinary IPStorm/IPStorm
binary is Mach-O
...
segment (size: 0) __PAGEZERO's entropy: 0.000000
segment (size: 8216576) __TEXT's entropy: 7.884009
segment (size: 16) __LINKEDIT's entropy: 0.000000

total compressed data: 8216576.000000
total compressed data vs. size: 0.999998

binary appears to be packed
significant amount of high-entropy data detected
```

Hooray! The code correctly detected that the malware is packed. This is because the `__TEXT` segment has a very high entropy (7.884 out of 8), and because it's the only segment containing any data, the ratio of packed data to the overall binary length is very high. Equally important is the fact that the code correctly determined that an unpacked version of the malware is indeed no longer packed:

```
% ./parseBinary IPStorm/IPStorm_unpacked
binary is Mach-O
...
segment (size: 0) __PAGEZERO's entropy: 0.000000
segment (size: 17190912) __TEXT's entropy: 6.185554
segment (size: 1265664) __DATA's entropy: 5.337738
segment (size: 1716348) __LINKEDIT's entropy: 5.618924

total compressed data: 0.000000
total compressed data vs. size: 0.000000

binary does *not* appear to be packed
no significant amount of high-entropy data detected
```

In this unpacked binary, the tool detects more segments, but all have an entropy of around 6 or below. Thus, it doesn't classify any of them as containing compressed data, so the ratio of compressed data to binary size is zero.

As you've seen, this entropy-based approach can generically detect almost any packed binary, regardless of the packer used. This holds true even in the case of OceanLotus, whose authors used a customized version of UPX in an attempt to avoid detection:

```
% ./parseBinary OceanLotus.H/flashlightd
...
segment (size: 0) __PAGEZERO's entropy: 0.000000
segment (size: 45056) __TEXT's entropy: 7.527715
segment (size: 2888) __LINKEDIT's entropy: 6.201859
```

```
total compressed data: 45056.000000
total compressed data vs. size: 0.939763

binary appears to be packed
significant amount of high-entropy data detected
```

Although the packed malware doesn't contain any segments or sections that match known packers, the large __TEXT segment contains a very high amount of entropy (7.5+). As such, the code correctly determines that the OceanLotus sample is packed.

Detecting Encrypted Binaries

While Apple encrypts the Intel versions of various system binaries, encrypted third-party binaries are rarely legitimate, and you should flag these for closer analysis. *Binary encryptors* encrypt the original malware code at the binary level. To automatically decrypt the malware at runtime, the encryptor will often insert a decryption stub and keying information at the start of the binary unless the operating system natively supports encrypted binaries, which macOS does.

As with packed binaries, we can detect encrypted binaries using entropy calculations, as any well-encrypted file will have a very high level of randomness. Thus, the code provided in the previous section should identify them. However, you might find it worthwhile to write code that focuses specifically on detecting binaries encrypted with the native macOS encryption scheme. The encryption scheme is undocumented and proprietary, so any third-party binary leveraging it should be treated as suspect.

We can see in the open source macOS Mach-O loader[12] how to detect such binaries. In the loader's code, we find mention of an LC_SEGMENT_64 flag value named SG_PROTECTED_VERSION_1 whose value is 0x8. As explained in Apple's *mach-o/loader.h* file, this means the segment is encrypted with Apple's proprietary encryption scheme:

```
#define SG_PROTECTED_VERSION_1  0x8 /* This segment is protected.  If the
                                       segment starts at file offset 0, the
                                       first page of the segment is not
                                       protected.  All other pages of the
                                       segment are protected. */
```

Usually, malware will encrypt only the __TEXT segment, which contains the binary's executable code.

Although it's rare to discover malware leveraging this proprietary encryption scheme, we find an example in a HackingTeam implant installer. Using otool, let's dump the load commands of this binary. Sure enough, the flags of the __TEXT segment are set to SG_PROTECTED_VERSION_1 (0x8):

```
% otool -l HackingTeam/installer
...
Load command 1
      cmd LC_SEGMENT
```

```
  cmdsize 328
  segname __TEXT
   vmaddr 0x00001000
   vmsize 0x00004000
  fileoff 0
 filesize 16384
  maxprot 0x00000007
 initprot 0x00000005
   nsects 4
    flags 0x8
```

To detect if a binary is encrypted using this native encryption scheme, we can simply iterate over its `LC_SEGMENT_64` load commands, looking for any that have the `SG_PROTECTED_VERSION_1` bits set in the `flags` member of the `segment_command_64` structure (Listing 2-20).

```
if(SG_PROTECTED_VERSION_1 == (segment->flags & SG_PROTECTED_VERSION_1)) {
    // Segment is encrypted.
    // Add code here to report this or to perform further processing.
}
```

Listing 2-20: Checking whether a segment is encrypted with the native macOS encryption scheme

This chapter has focused on 64-bit Mach-Os, but the HackingTeam installer is almost 10 years old and was distributed as a 32-bit Intel binary, which isn't compatible with recent versions of macOS. To write code capable of detecting HackingTeam's 32-bit installer, we'd have to make sure it uses the 32-bit versions of the Mach-O structures, such as `mach_header` and `LC_SEGMENT`.[13] If we made these changes and ran the code against the installer, it would correctly flag the binary as leveraging Apple's proprietary encryption scheme:

```
% ./parseBinary HackingTeam/installer
...
segment __TEXT's flags: 'SG_PROTECTED_VERSION_1'

binary is encrypted
```

We noted that though macOS does natively support encrypted binaries, because this is not documented, any third-party binary that is encrypted in this manner should be closely examined, as it may be malware with something to hide.[14]

Conclusion

In this chapter, you learned how to confirm that a file is a Mach-O or a universal binary containing Mach-Os. Then you extracted dependencies and names and detected whether the binary was packed or encrypted.

Of course, there are many other interesting things you could do with a Mach-O binary to classify it as benign or malicious. Take a look at Kimo Bumanglag's Objective by the Sea talk for ideas.[15]

A final thought: I've noted that no single data point covered in this chapter can definitively indicate that a binary is malicious. For example, nothing stops legitimate developers from packing their binaries. Luckily, we have another powerful mechanism at our disposal to detect malware: code signing. Chapter 3 is dedicated to this topic. Read on!

Notes

1. UniqMartin, comment on "FatArch64," Homebrew, July 7, 2018, *https://github.com/Homebrew/ruby-macho/issues/101#issuecomment-403202114.*
2. "magic," Apple Developer Documentation, *https://developer.apple.com/documentation/kernel/fat_header/1558632-magic.*
3. See *utils.cpp* at *https://github.com/apple-oss-distributions/dyld/blob/d1a0f6869ece370913a3f749617e457f3b4cd7c4/libdyld/utils.cpp.*
4. Patrick Wardle, "Apple Gets an 'F' for Slicing Apples," Objective-See, February 22, 2024, *https://objective-see.org/blog/blog_0x80.html.*
5. For more on universal binaries, see Howard Oakley, "Universal Binaries: Inside Fat Headers," *The Eclectic Light Company*, July 28, 2020, *https://eclecticlight.co/2020/07/28/universal-binaries-inside-fat-headers/.*
6. Patrick Wardle, "Burned by Fire(fox)," Objective-See, June 23, 2019, *https://objective-see.org/blog/blog_0x45.html.*
7. For more details on ZuRu, see Patrick Wardle, "Made in China: OSX.ZuRu," Objective-See, September 14, 2021, *https://objective-see.org/blog/blog_0x66.html.*
8. See *https://upx.github.io.*
9. "Entropy (information theory)," Wikipedia, *https://en.wikipedia.org/wiki/Entropy_(information_theory).*
10. To gain a deeper understanding of entropy, see Ms Aerin, "The Intuition Behind Shannon's Entropy," Towards Data Science, September 30, 2018, *https://towardsdatascience.com/the-intuition-behind-shannons-entropy-e74820fe9800.*
11. See *https://github.com/hiddenillusion/AnalyzePE/blob/master/peutils.py* and *https://github.com/erocarrera/pefile/blob/master/pefile.py.*
12. See *https://opensource.apple.com/source/xnu/xnu-7195.81.3/EXTERNAL_HEADERS/mach-o/loader.h.*
13. For more details about HackingTeam's encrypted installer, see Patrick Wardle, "HackingTeam Reborn; A Brief Analysis of an RCS Implant

Installer," Objective-See, February 26, 2016, *https://objective-see.org/blog/blog_0x0D.html.*

14. You can read more about the macOS support of encrypted binaries and how to decrypt them in Patrick Wardle, *The Art of Mac Malware: The Guide to Analyzing Malicious Software*, Volume 1 (San Francisco: No Starch Press, 2022), 187–218, or in Amit Singh, "'TPM DRM' in Mac OS X: A Myth That Won't Die," *OSX Book*, December 2007, *https://web.archive.org/web/20200603015401/http://osxbook.com/book/bonus/chapter7/tpmdrmmyth/.*

15. Kimo Bumanglag, "Learning How to Machine Learn," paper presented at Objective by the Sea v5, Spain, October 6, 2022, *https://objectivebythesea.org/v5/talks/OBTS_v5_kBumanglag.pdf.* To learn more about the Mach-O format in general, consult Wardle, *The Art of Mac Malware*, 1:99–123; Bartosz Olszanowski, "Mach-O Reader - Parsing Mach-O Headers," *Olszanowski Blog*, May 8, 2020, *https://olszanowski.blog/posts/macho-reader-parsing-headers/*; and Alex Denisov, "Parsing Mach-O Files," *Low Level Bits*, August 20, 2015, *https://lowlevelbits.org/parsing-mach-o-files/.*