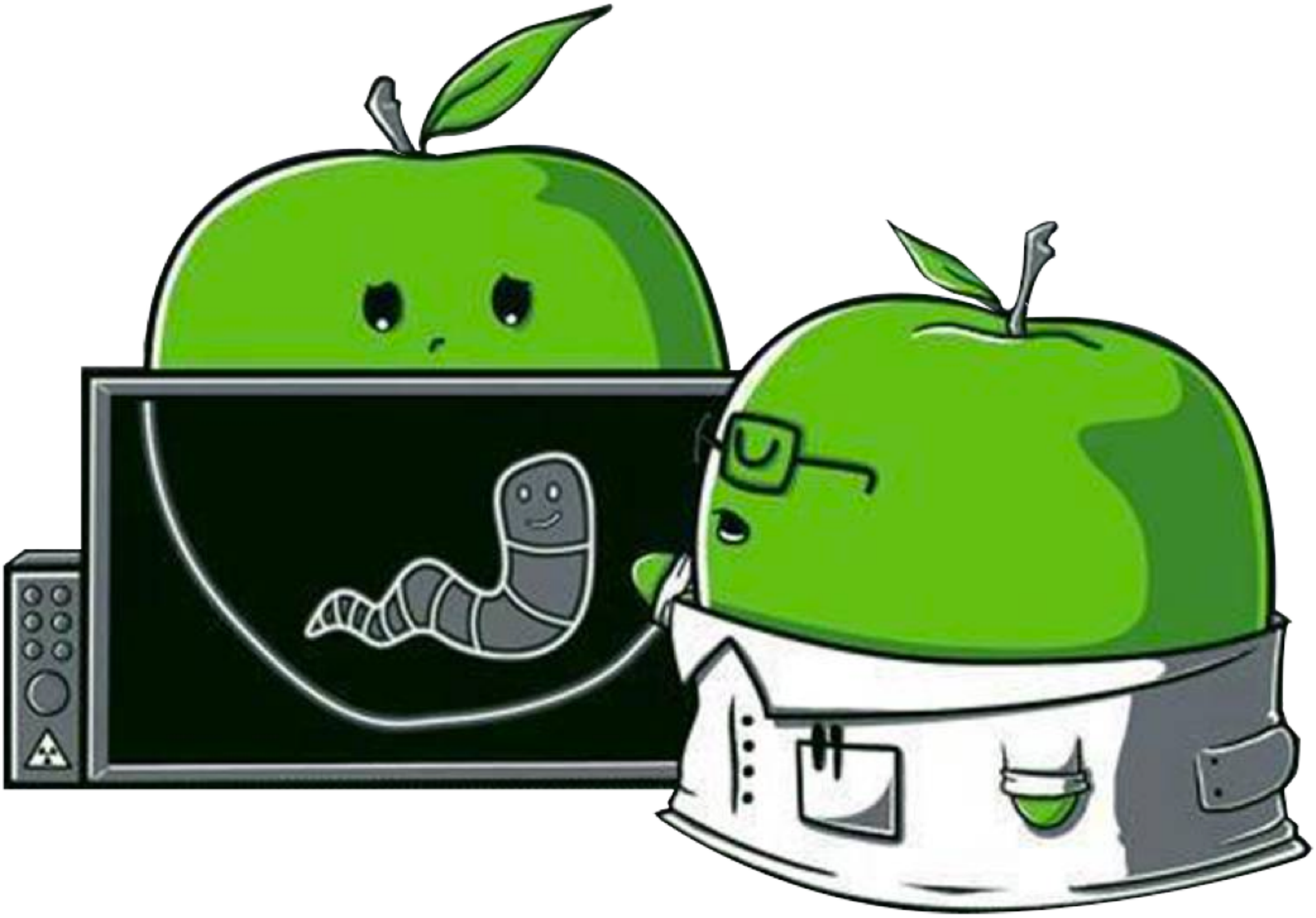


Chapter 0x6: Binary Triage

📝 Note:

This book is a work in progress.

You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!

To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

Apple notes that Mach-O, (shorthand for “Mach object file format”), *“is the native executable format of binaries in OS X and is the preferred format for shipping code.”* [1]

As the majority of Mac malware is compiled into and distributed as Mach-O binaries, it is important to have a solid understanding of this file format.

```
$ file Final_Presentation.app/Contents/MacOS/usrnode

Final_Presentation.app/Contents/MacOS/usrnode: Mach-O 64-bit executable x86_64
```

a 64-bit Mach-O executable
(OSX.WindTail)

Unfortunately, as Mach-O is a binary file format, analyzing and understanding such files requires specific analysis tools. Tools that often culminate with a disassembler.

📝 Note:

For the definitive guide on Mach-O binaries, see Apple’s documentation:

“OS X ABI Mach-O File Format Reference” [1]

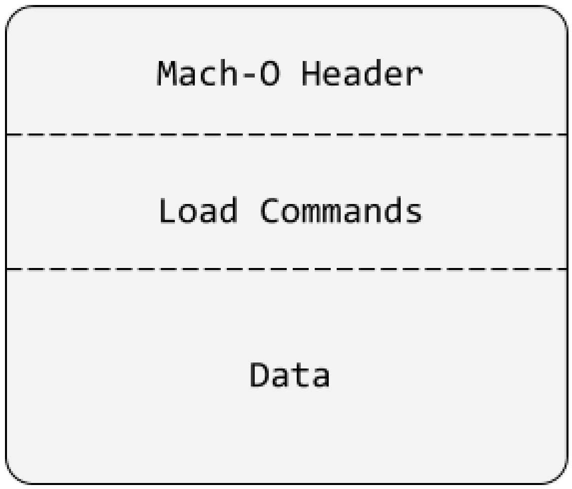
Executable binary file formats are rather complex, and the Mach-O file format is no exception. The good news is that one only needs an elementary understanding of the Mach-O file format and several related concepts for malware analysis purposes.

📝 Note:

For the interested reader, an in-depth, and frankly quite excellent, writeup on the Mach-O file format can be found here:

"Parsing Mach-O File" [2]

At a basic level, a Mach-O file consists of three sequential parts, or regions: a header, load commands, and data.



Mach-O Header

Mach-O files start with a Mach-O header:

> *"At the beginning of every Mach-O file is a header structure that identifies the file as a Mach-O file. The header also contains other basic file type information, indicates the target architecture, and contains flags specifying options that affect the interpretation of the rest of the file."* [1]

A Mach-O header is a structure of type `mach_header_64` (or 32-bit `mach_header`), defined in `mach-o/loader.h`:

```
struct mach_header_64 {
        uint32_t        magic;          /* mach magic number identifier */
        cpu_type_t      cputype;        /* cpu specifier */
        cpu_subtype_t   cpusubtype;     /* machine specifier */
        uint32_t        filetype;       /* type of file */
        uint32_t        ncmds;          /* number of load commands */
```

```
        uint32_t        sizeofcmds;     /* the size of all the load commands */
        uint32_t        flags;          /* flags */
        uint32_t        reserved;       /* reserved */
};
```

mach_header_64 structure
(mach-o/loader.h)

Apple's comments in the `loader.h` file should provide a sufficient, albeit succinct, description of each member (within the `mach_header_64` structure).

Of particular note is the `filetype` member, which describes the type of file. Several possible values include (from `mach-o/loader.h`):

- `MH_EXECUTE` (0x2)
  Standard Mach-O executable

- `MH_DYLIB` (0x6)
  A Mach-O dynamic linked library (i.e. .dylib)

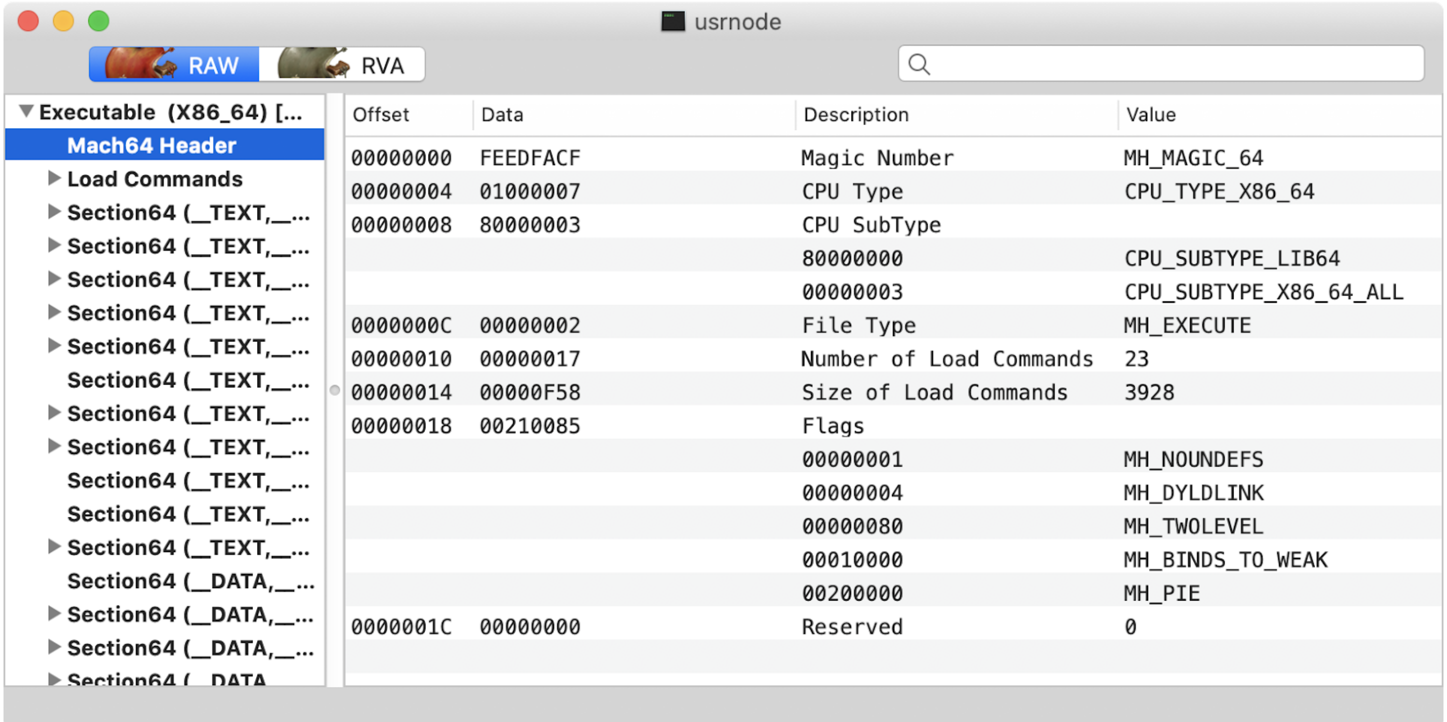
- `MH_BUNDLE` (0x8)
  A Mach-O bundle (i.e. .bundle)

To dump, or parse, the contents of a Mach-O file one can make use of the `/usr/bin/otool` utility. For example, to dump the Mach-O header, execute `otool` with the `-hv` flags:

```
$ otool -hv Final_Presentation.app/Contents/MacOS/usrnode

Mach header
   magic         cputype      cpusubtype       filetype      ncmds       sizeofcmds
MH_MAGIC_64       X86_64          ALL           EXECUTE        23           3928
```

Dumping OSX.WindTail's Mach-O header
(via `otool`)

Or, if you prefer a UI, MachOView [3] is a lovely utility!



Dumping a Mach-O header
(via MachOView)

📝 Note:

Apple notes that a "*Mach-O file contains code and data for one architecture.*" [1]

In order to create a single binary that can execute on systems with different architectures (i.e. 32-bit, 64-bit, etc.), multiple Mach-O binaries can be wrapped in a universal (or "fat") binary.

Such binaries start with a header (type: fat_header), then the architecture-specific Mach-O binaries concatenated together.

One can dump the fat_header via: otool -fv

Mach-O Load Commands

Following the Mach-O header are the binary's load commands, which instruct ("command") the dynamic loader (dyld) how to, well, load (and layout) the binary in memory.

"Directly following the header are a series of variable-size load commands that specify the layout and linkage characteristics of the file. Among other information, the load commands can specify:

- *The initial layout of the file in virtual memory*
- *The location of the symbol table (used for dynamic linking)*
- *The initial execution state of the main thread of the program*
- *The names of shared libraries that contain definitions for the main executable's imported symbols"* [1]

A Mach-O binary's load commands can be viewed via the `otool`, using the `-l` flag:

```
$ otool -l Final_Presentation.app/Contents/MacOS/usrnode
…

Load command 0
      cmd LC_SEGMENT_64
  cmdsize 72
  segname __PAGEZERO
   vmaddr 0x0000000000000000
   vmsize 0x0000000100000000
  fileoff 0
 filesize 0
  maxprot 0x00000000
 initprot 0x00000000
   nsects 0
    flags 0x0
Load command 1
      cmd LC_SEGMENT_64
  cmdsize 952
  segname __TEXT
   vmaddr 0x0000000100000000
   vmsize 0x0000000000013000
  fileoff 0
 filesize 77824
  maxprot 0x00000007
 initprot 0x00000005
   nsects 11
    flags 0x0

...
```

Dumping `OSX.WindTail`'s load commands (via `otool`)

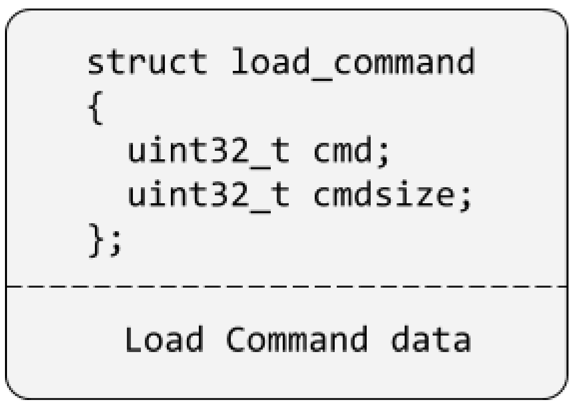
We’re aiming to gain a foundational understanding of the Mach-O file format for the purpose of malware analysis, so we won’t cover all supported load commands. However, several are quite pertinent.

Load commands all begin with a `load_command` structure, defined in `mach-o/loader.h`:

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

load_command structure
(mach-o/loader.h)

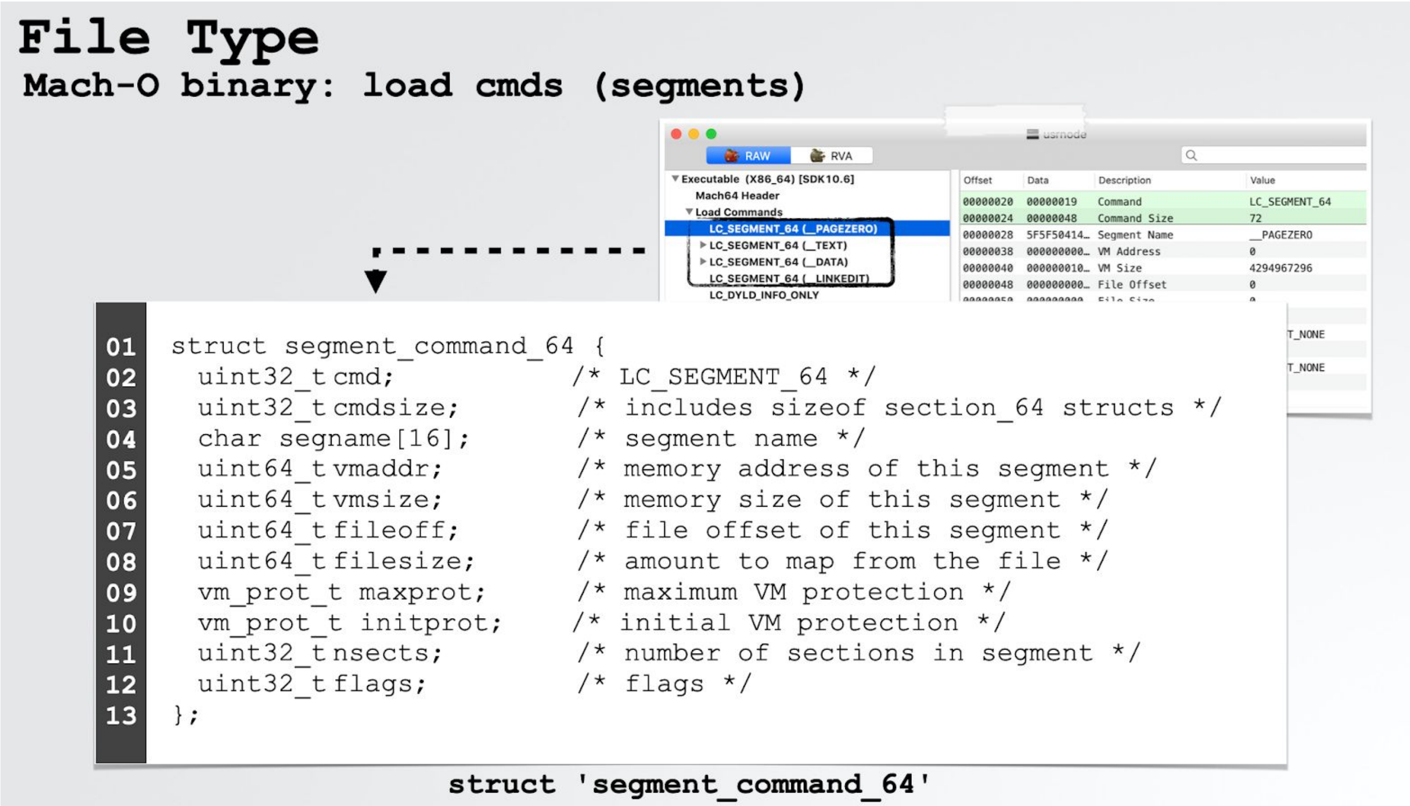
Here, `load_command.cmd` describes the type of load command, while the size of the load command is specified in `load_command.cmdsize`. Note that the load command’s data follows immediately after the `load_command` structure, and such data is specific to the type of the load command:



A common type of load command is `LC_SEGMENT/LC_SEGMENT_64`, which describes a segment. Apple defines a segment in the following manner:

> *“A segment defines a range of bytes in a Mach-O file and the addresses and memory protection attributes at which those bytes are mapped into virtual memory when the dynamic linker loads the application.”* [1]

As shown in the following image, `LC_SEGMENT/LC_SEGMENT_64` load commands contain all the relevant information for the dynamic loader (`dyld`) to map the segment into memory (and set its memory permissions):



`LC_SEGMENT/LC_SEGMENT_64` load command

Several segments you'll likely encounter while analyzing Mach-O binaries include:

- `__TEXT` segment
  Contains executable code and data that is read-only

- `__DATA` segment
  Contains data that is writable

- `__LINKEDIT` segment
  Contains information for the linker (`dyld`) such as, "*symbol, string, and relocation table entries*." [1]

If the binary was written in objective-C, it may have an `__OBJC` segment that contains information used by the Objective-C runtime. Though this information might also be found in the `__DATA` segment, within various in `__objc_*` sections.

> 📝 Note:
>
> Segments can contain multiple sections (each section containing code or data of the same types). More on sections below...

Once a binary is loaded into memory (by the dynamic linker/loader dyld), execution begins at the binary's entry point. How does the dyld locate said entry point? Via the LC_MAIN load command!

This load command is (cumulatively) a structure of type entry_point_command:

```
struct entry_point_command {
    uint32_t  cmd;      /* LC_MAIN only used in MH_EXECUTE filetypes */
    uint32_t  cmdsize;  /* 24 */
    uint64_t  entryoff; /* file (__TEXT) offset of main() */
    uint64_t  stacksize;/* if not zero, initial stack size */
};
```

LC_MAIN's entry_point_command structure
(mach-o/loader.h)

The most important member of the LC_MAIN load command is the entryoff, which contains the offset of the binary's entry point. At load time, dyld simply adds this value to the (in-memory) base of the binary, then jumps to this instruction to kickoff execution of the binary's code.

> "*LC_MAIN gives the address of the entry point (main()) and [the loader] dyld jumps right to that...*" [4]

📝 Note:

The LC_MAIN load command replaces the deprecated LC_UNIXTHREAD load command.

If you're analyzing older Mach-O binaries, you may still come across the LC_UNIXTHREAD, which contains the entire context (read: register values) of the initial thread. The **EIP/RIP** register in this context contains the address of the binary's initial entry point.

📝 Note:

A Mach-O binary can contain one or more constructors, that will be executed ***before*** the address specified in LC_MAIN.

The offsets of any constructors are held in the __mod_init_func section of the __DATA_CONST segment.

More on this topic shortly, but be aware when analyzing Mac malware that execution may begin within such a constructor, prior to the binary's main entry point (LC_MAIN).

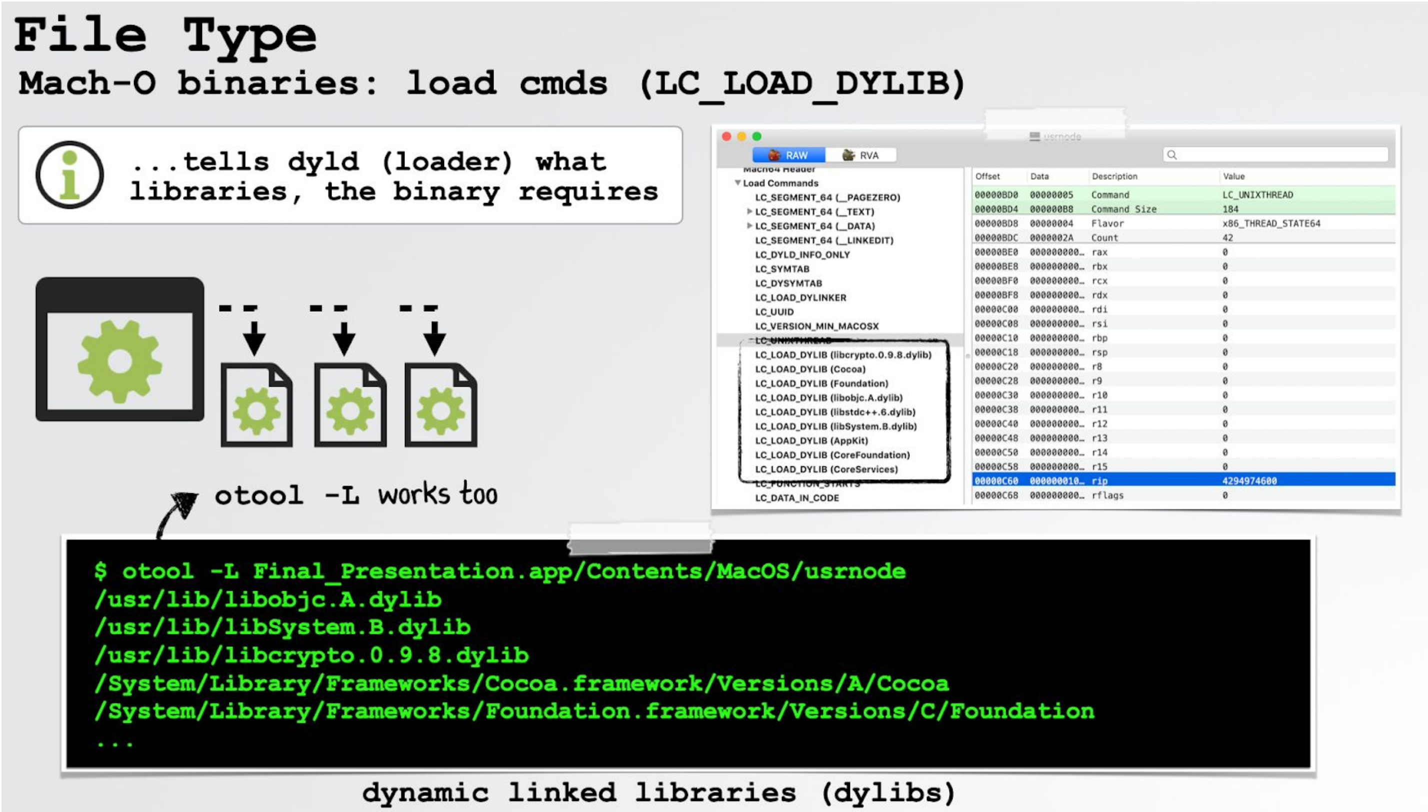
When analyzing Mac malware, another relevant load command type is `LC_LOAD_DYLIB`. In short, the `LC_LOAD_DYLIB` load command describes a dynamic library dependency which instructs the loader (`dyld`) to load and link said library. There is a `LC_LOAD_DYLIB` load command for each library that the Mach-O binary requires (i.e. has a dependency on).

This load command is (cumulatively) a structure of type `dylib_command` (which contains a `struct dylib`, describing the actual dependent dynamic library):

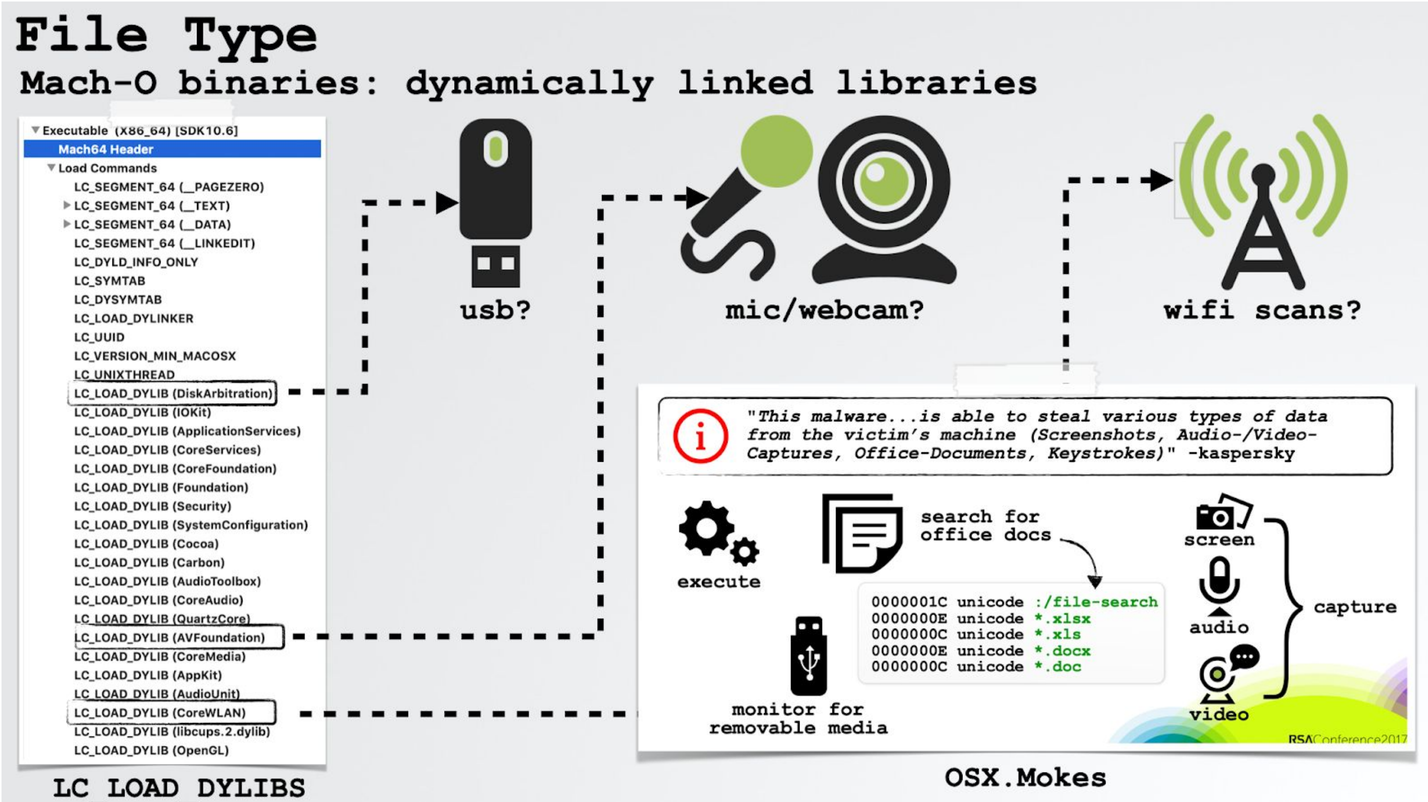
```
struct dylib_command {
        uint32_t        cmd;            /* LC_LOAD_{,WEAK_}DYLIB */
        uint32_t        cmdsize;        /* includes pathname string */
        struct dylib    dylib;          /* the library identification */
};

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*/
};
```

`LC_LOAD_DYLIB`'s `dylib_command` & `dylib` structures
(`mach-o/loader.h`)

To parse a Mach-O binary's `LC_LOAD_DYLIB` load command to view the binary's dependencies, use the `otool` utility, with the `-L` flag. Or, MachOView [3] works as well.



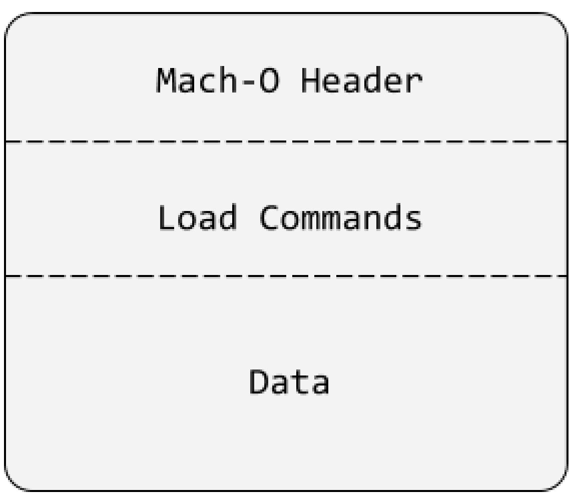
From a malware analysis point of view, a binary's `LC_LOAD_DYLIB` load commands can shed insight into the capabilities of malware. For example, a binary that contains a `LC_LOAD_DYLIB` load command that references the `DiskArbitration` library may be interested in monitoring USB drives (perhaps to exfil files off such drives). A dependency on the `AVFoundation` library may indicate that the malware seeks to capture audio and video from infected systems.



Ascertaining capabilities via `LC_LOAD_DYLIB` load commands (`OSX.Mokes`)

Mach-O Data (Segments)

Recall the following diagram representing the (basic) structure of a Mach-O binary:



Following the Load Commands is the rest of the Mach-O binary, largely consisting of the actual binary code. Such data is organized into segments, described by `LC_SEGMENT/LC_SEGMENT_64` Load Commands, which can contain multiple sections. As Apple notes, each section contains code or data of the same type:

> *"A Mach-O binary is organized into segments. Each segment contains one or more sections. Code or data of different types goes into each section."* [5]

For example, the `_TEXT` segment contains executable code and data that is read-only. Common sections within this segment may include:

- `__text`
  Compiled binary code

- `__const`
  Constant data

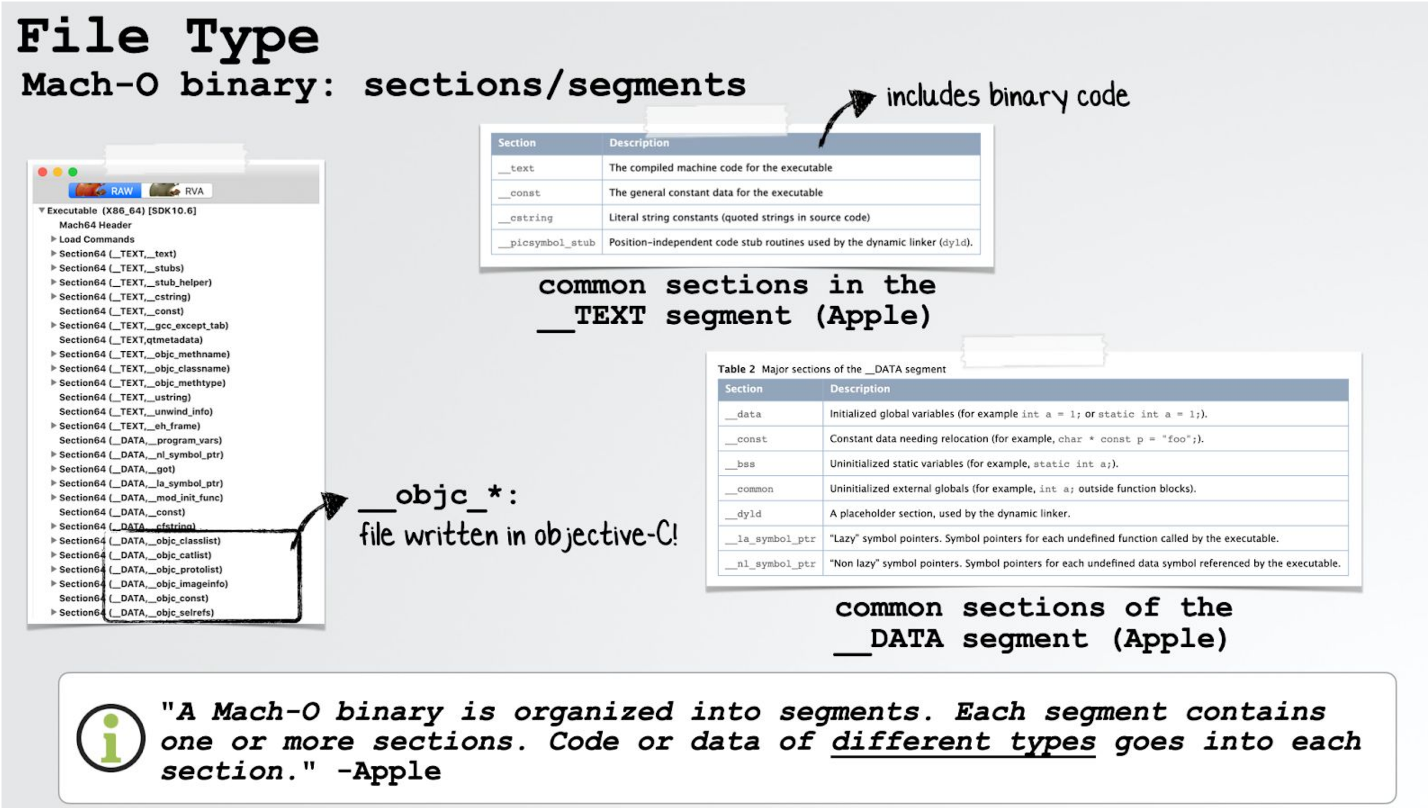
- `__cstring`
  String constants

The `__DATA` segment contains data that is writable. A few of the (more common) sections within this segment may include:

- `__data`
  Global variables (that have been initialized)

- `__bss`
  Static variables (that have not been initialized)

- `__objc_*` (`__objc_classlist`, `__objc_protolist`, etc)
  Information used by the Objective-C runtime



Mach-O sections/segments

With an elementary understanding of the Mach-O file format, let’s now focus our attention on tools and techniques that aim to answer the question forever faced by malware analysts: “is this (Mach-O) binary malicious!?”

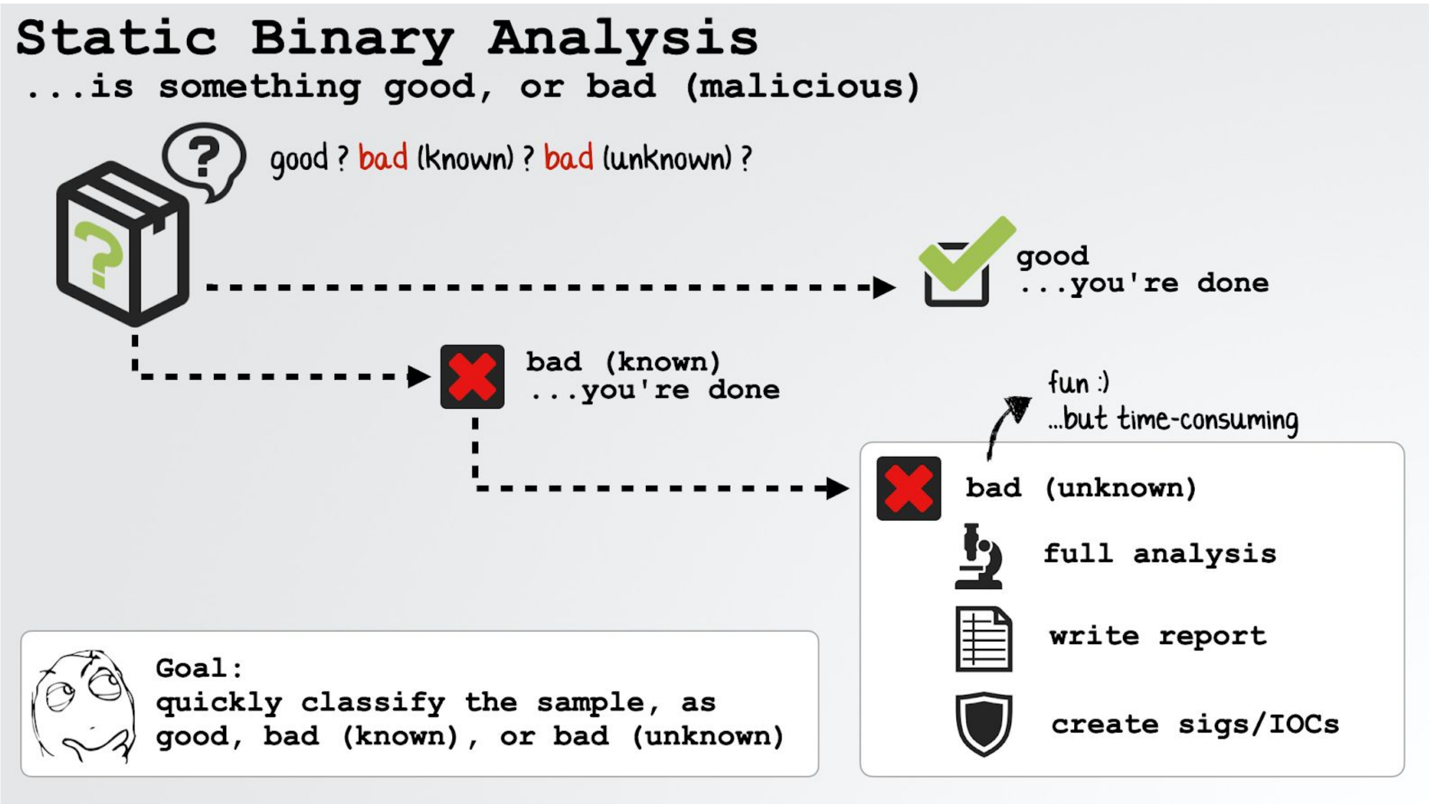
Static Analysis of Mach-O Files

Generally speaking, the goal of malware analysis is to classify a sample as benign, malicious (but known), or malicious (and previously unknown).

If a sample turns out to be benign, hooray you’re done! ...generally no point (from a malware analysts point of view) to continuing analyzing a legitimate and benign piece of software.

If a sample is malicious, but is a known malware sample, (unless you’re analyzing the sample for educational purposes), you’re done as well. It’s likely that analysis reports and indicators of compromise (IoCs) have already been created for the sample.

However, if you determine the sample is malicious and appears to either be a new variant, or an entirely new specimen, well, you're not done - yet! Such samples generally require a full analysis and report, as well as the creation of IoCs.



A key point is to classify samples efficiently. As, speaking from personal experience, spending several days analyzing a sample only to find out it is a well known piece of malware can be frustrating. Though of course, the educational experience of such a process has its merits.

Due to their readability, it is often quite trivial to classify scripts, and other non-binary file formats, as benign or malicious. However, binary file formats (read: Mach-O) require a myriad of tools to both classify and comprehensively analyze.

As such, let's now dive into the static analysis of Mach-O binaries.

As noted, static analysis of Mach-O binaries generally requires tools. Such tools generally have some understanding of the Mach-O file format, though more elementary ones may be file type agnostic.

Also, recall our goal to efficiently classify a binary as benign or malicious and, for malicious binaries, identify it as an already known sample.

To accomplish this, we'll start by extracting and analyzing various file attributes, such as:

- Hashes
- Code-signing information
- Embedded strings

If one cannot ascertain if a sample is benign or malicious via these elementary tools and techniques, more comprehensive tools may be required (such as a disassembler ...covered in the next chapter).

Hashes

One of the simplest ways to determine if a Mach-O binary is known, and thus has already been classified as benign or malicious, is to simply compute and look up its hash online.

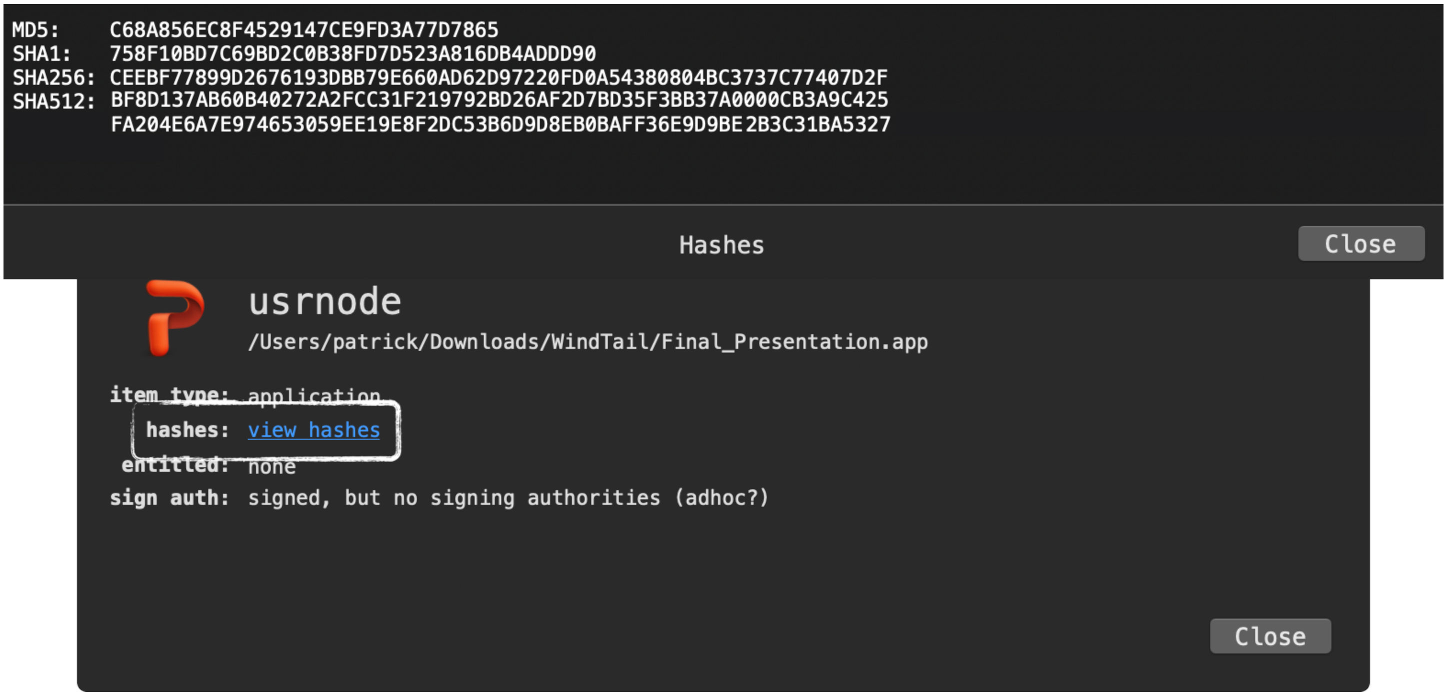
Hashing algorithms, such as MD5 and SHA-*, are most commonly used in public file repositories of online malware collections. Luckily, macOS ships with built-in utilities for computing such hashes (`/sbin/md5` and `/usr/bin/shasum`).

Here, we generate both the MD5 and SHA-1 hash of Mach-O binary (`usrnode`) found within a suspicious application bundle:

```
$ md5 Final_Presentation.app/Contents/MacOS/usrnode
MD5 (usrnode) = c68a856ec8f4529147ce9fd3a77d7865

$ shasum -a 1 Final_Presentation.app/Contents/MacOS/usrnode
758f10bd7c69bd2c0b38fd7d523a816db4addd90  usrnode
```

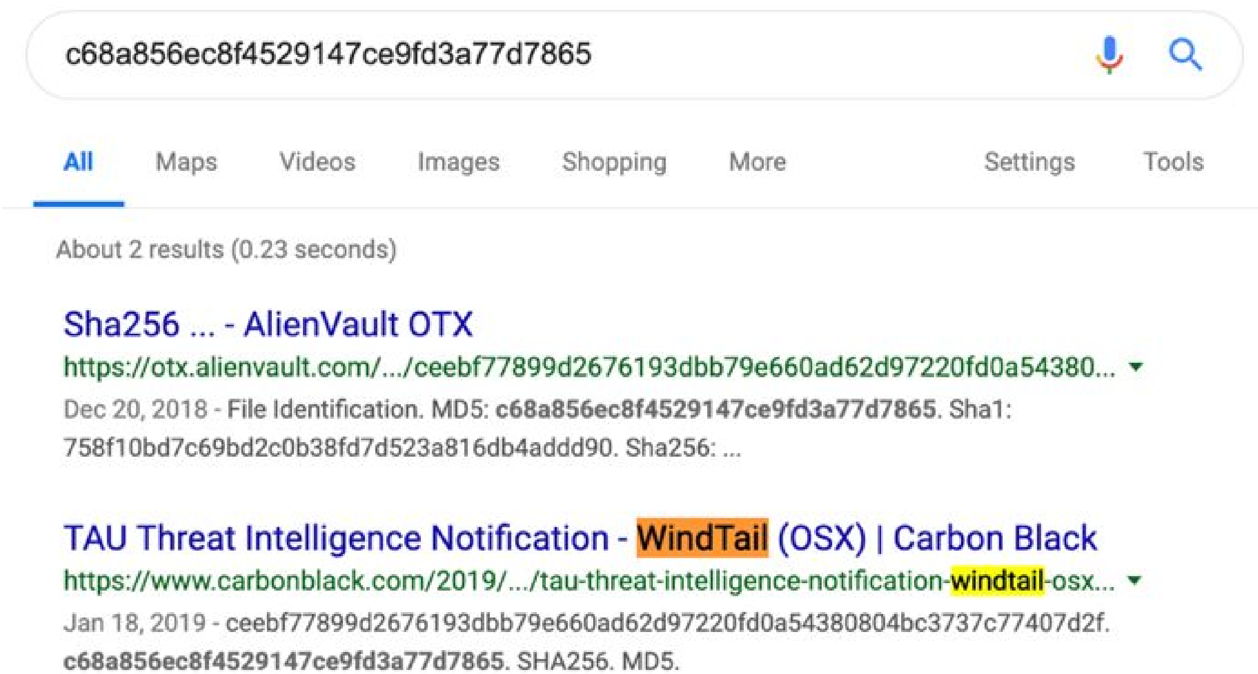
Hashing

If you're more comfortable using a UI utility, the [WhatsYourSign](#) tool [6] (created by yours truly), will compute `MD5`, `SHA-1` `-256`, and `-512` hashes of files:

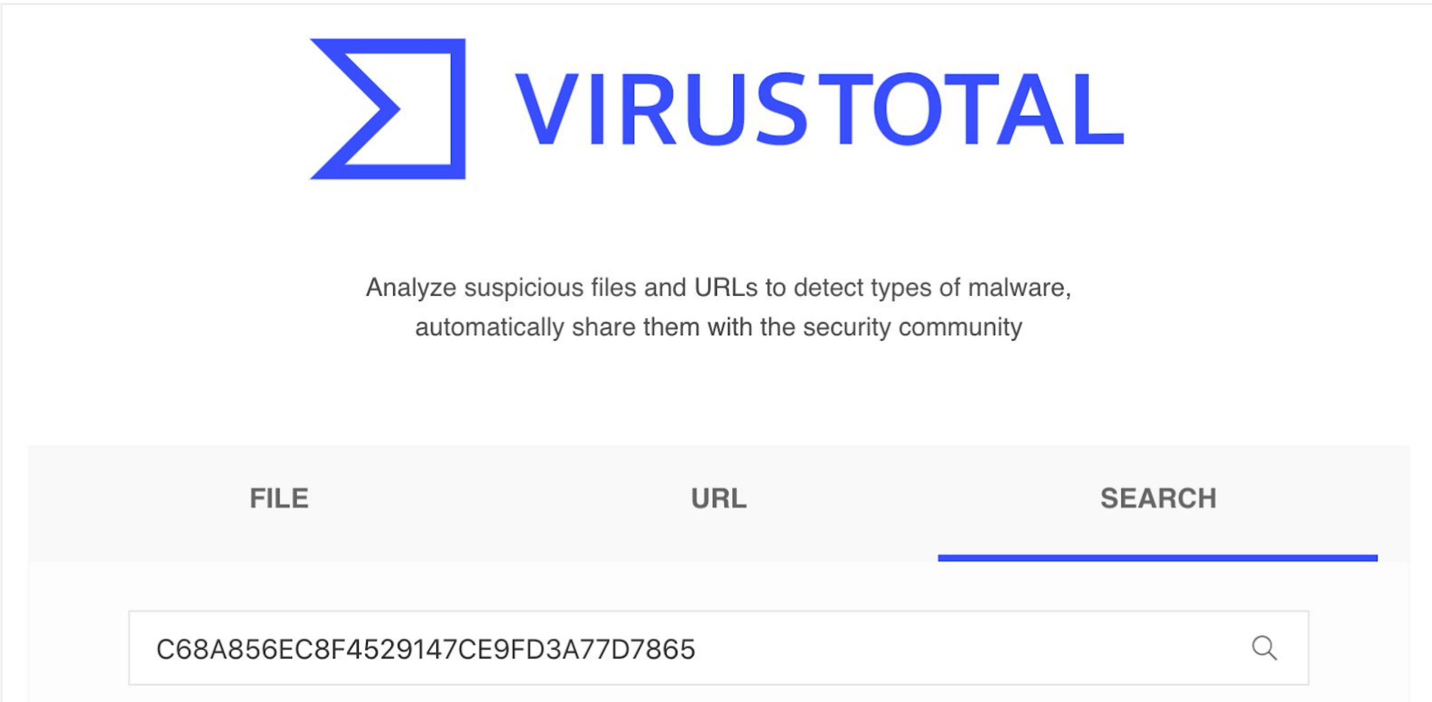


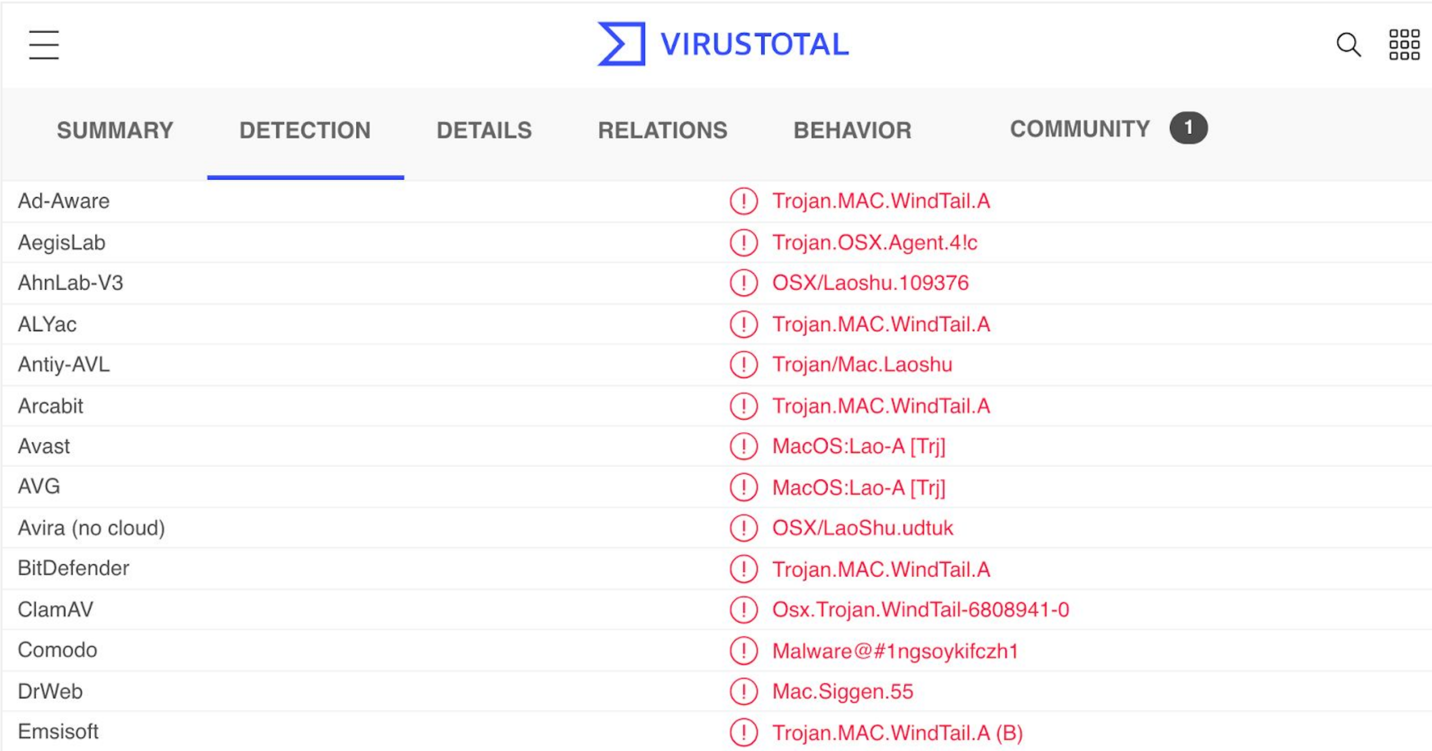
WhatsYourSign tool [6]

Googling the (MD5) hash, `C68A856EC8F4529147CE9FD3A77D7865`, readily identifies this binary as `OSX.WindTail`:



Searching for this same hash on VirusTotal [7], a free online antivirus “scanning portal” with a large collection of scan results, confirms this identification as well:





C68A856EC8F4529147CE9FD3A77D7865 -> OSX.WindTail
(VirusTotal)

If our goal was to simply classify the binary (usrnode) as benign or malicious, and if malicious, attempt to identify the sample, we've just accomplished this goal! ...simply via the binary's hash.

> 📝 Note:
>
> Hashes are a great way to conclusively match two binaries. For example, matching an unknown binary with a piece of legitimate software, or a known malware sample.
>
> However, hashes are quite 'brittle' as any file change will result in a completely different hash. As such, if a malware author modifies even a single bit there may be zero hash matches.
>
> Thus, hashing should be seen as a technique to identify known files that may have already been classified as benign or malicious. However, if no hash match is found, this should not be used as a metric to classify the file's nature. Other analysis tools and techniques should be leveraged.

Code Signing Information

Due to various Apple efforts, such as file quarantine, notarization, etc, the majority of software on macOS is signed. Such signing information may include:

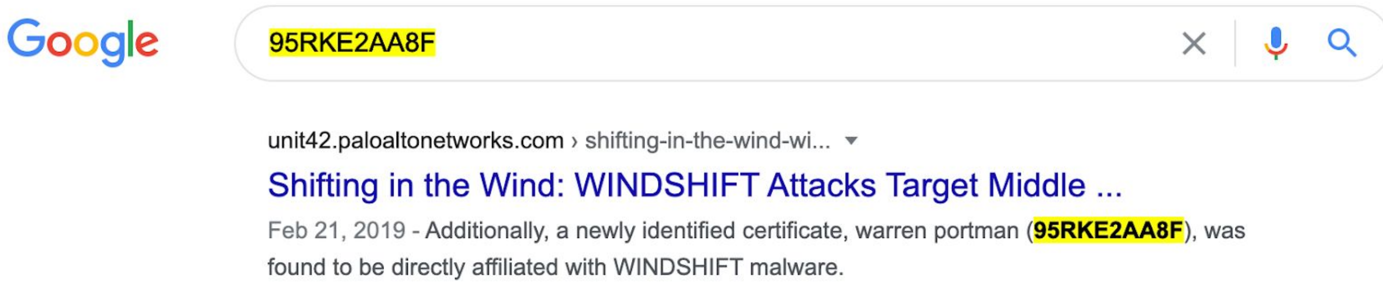
- Code-signing identifier
- Code-signing authorities
- Team identifier

As Apple notes, this allows one to confirm that a binary *"is from a known source and [it] hasn't been modified since it was last signed."* [8]

By extracting the code-signing information of (signed) Mach-O binaries, one may be able to quickly ascertain that an unknown binary is benign, or in some cases match it with known malware or malware creator. For example, if you are analyzing an unknown binary, and it is signed by Apple proper, rest assured, that binary is not malicious! On the other hand, if a binary is unsigned, or claims to be from a well established company but isn't signed by said company, this may be cause for further analysis.

Trojanized Firefox
(OSX/CreativeUpdater) [9]

Like hashes, code-signing information can also be used to find file matches online, and in some cases matching unknown files to known malware. For example, searching for the aforementioned `usrnode` binary's code-signing Team Identifier, `95RKE2AA8F`, quickly leads us to a match identifying it as a (known) sample associated with the WINDSHIFT malware family (specifically `OSX.WindTail`):



Team Identifier
(OSX.WindTail's)

Finally, if a Mach-O binary is signed, but its certificate has been revoked (by Apple), this is a red flag and likely indicates the binary is malicious.

Code signing information may be extracted from a Mach-O binary via Apple's `/usr/bin/codesign` utility (using the `-dvv` flags):

```
$ codesign -dvv Final_Presentation.app/Contents/MacOS/usrnode
Executable=Final_Presentation.app/Contents/MacOS/usrnode
Identifier=com.alis.tre
Format=app bundle with Mach-O thin (x86_64)

Authority=(unavailable)

TeamIdentifier=95RKE2AA8F
```

extracting a binary's code signing information
(`OSX.WindTail`)

This `OSX.WindTail` sample is signed, but has no signing authorities ('`Authority=(unavailable)`'). This indicates the sample is self-signed (ad hoc). Anecdotally speaking, self-signed binaries are rarely legitimate.

Looking at a legitimate Mach-O binary (Apple's built-in Calculator application), shows the full signing authority chain (Apple Root CA -> Apple Code Signing Certification Authority -> Software Signing):

```
$ codesign -dvv /System/Applications/Calculator.app/Contents/MacOS/Calculator
Executable=/System/Applications/Calculator.app/Contents/MacOS/Calculator
Identifier=com.apple.calculator
```

```
Format=app bundle with Mach-O thin (x86_64)

Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA

TeamIdentifier=not set
```

legitimately signed (Apple) application (`Calculator.app`)

A legitimate, signed 3rd-party application provides an example of a binary signed with an Apple Developer ID (note authority #2, "`Developer ID Certification Authority`"):

```
$ codesign -dvv KnockKnock.app/Contents/MacOS/KnockKnock
Executable=KnockKnock.app/Contents/MacOS/KnockKnock
Identifier=com.objective-see.KnockKnock
Format=app bundle with Mach-O thin (x86_64)

Authority=Developer ID Application: Objective-See, LLC (VBG97UB4TA)
Authority=Developer ID Certification Authority
Authority=Apple Root CA

TeamIdentifier=VBG97UB4TA
```

legitimately signed (3rd-party) application

Finally, `codesign` will simply display: "*code object is not signed at all*" for unsigned Mach-O binaries.

We noted earlier that if the code-signing certificate used to sign a Mach-O has been revoked, this may mean the binary was deemed (by Apple) to be malicious.

Using macOS's `/usr/sbin/spctl` utility, one can check the status of a binary's code-signing certificate. If a certificate has been revoked, the utility will display `CSSMERR_TP_CERT_REVOKED`:

```
$ spctl --assess Final_Presentation.app/Contents/MacOS/usrnode
Final_Presentation.app/Contents/MacOS/usrnode: CSSMERR_TP_CERT_REVOKED
```

revoked code-signing certificate (`CSSMERR_TP_CERT_REVOKED`) (`OSX.WindTail`)

The WhatsYourSign tool [6] can also be used to extract code-signing information from Mach-O binaries, albeit directly via the UI. Here, an OSX.WindTail specimen:

WhatsYourSign

📝 Note:

Code-signing is an important, albeit involved topic. Interested in learning more? See:

- Code Signing – Hashed Out [10]
- macOS Code Signing In Depth [11]

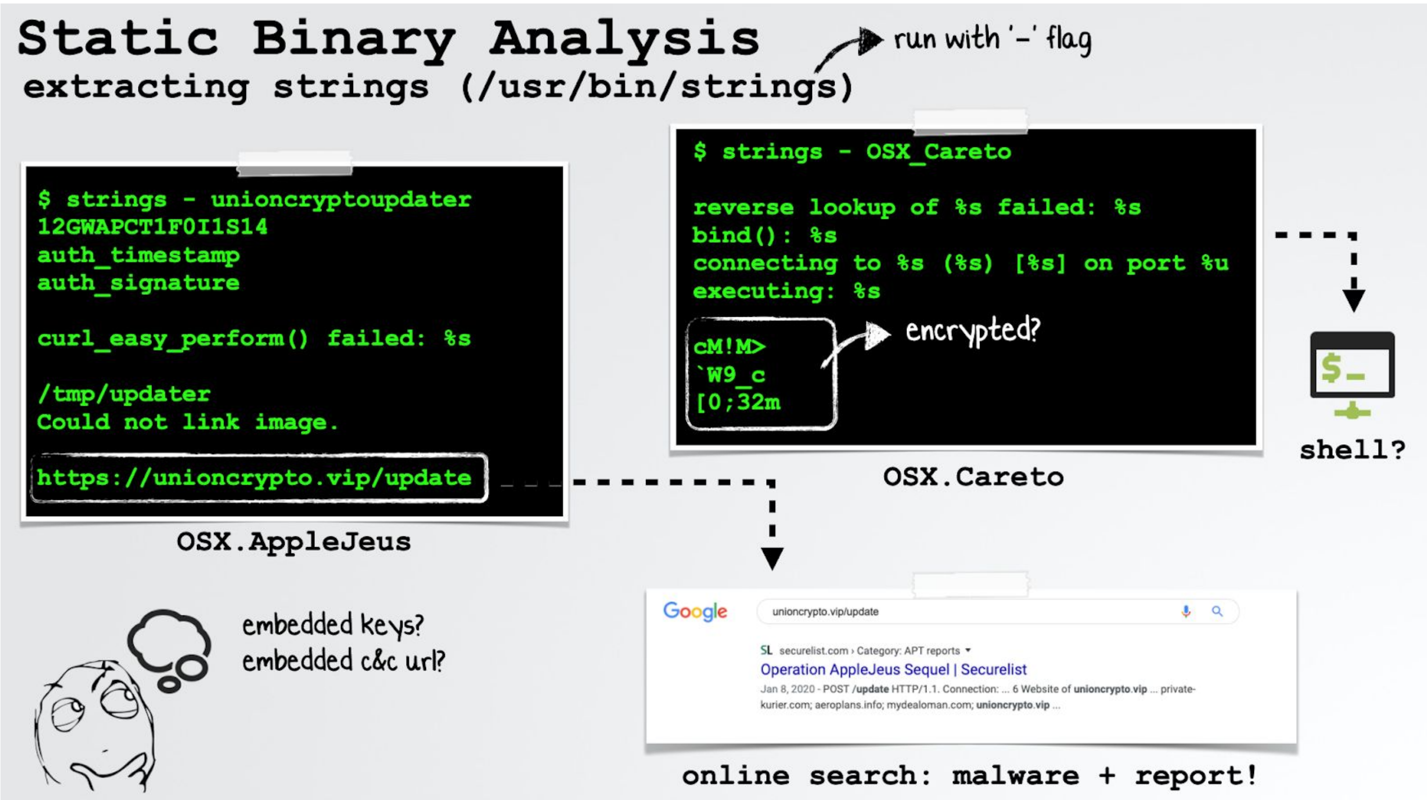
Strings

Though the Mach-O file format is a binary file format (i.e. not directly 'readable' by mere mortals), various non-binary data may still be found within it ...for example, strings (defined here as sequences of printable characters).

Using the aptly named /usr/bin/strings utility, one can extract strings from a compiled Mach-O binary. Such strings can include:

- debug or error messages
- method or function names
- configuration files and/or urls

...which can provide valuable insight into the capabilities of the binary being analyzed.



strings ...for the win!

```
$ man strings
NAME
       strings - find the printable strings in a object, or other binary, file

DESCRIPTION
       Strings looks for ASCII strings in a binary file or standard input.
A string is any sequence of 4 (the default) or more printing characters [ending at,
but not including, any other character or EOF].

Unless the - flag is given, strings looks in all sections of the object files except
the (__TEXT,__text) section.  If no files are specified standard input is read.
```

strings's man page

📝 Note:

When extracting strings from a binary, always run the strings utility with the "-" flag. As noted in its man page, this *"causes strings to look for strings in all bytes of the files."* [12] Otherwise, strings will only scan certain sections of the file!

Also, the strings utility only scans for ASCII strings, thus unicode strings may be missed! A 'unicode' aware utility (such as most disassemblers) can be used to extract such multi-character strings.

Finally, the string utility (by design) is fairly 'dumb,' in the sense that it simply displays sequences of printable characters. As such, many random sequences of binary values, that just happen to be printable, may be displayed. However, valid strings of interest should be easy to spot in the output.

Here, we run strings on a unknown Mach-O binary (`usrnode`):

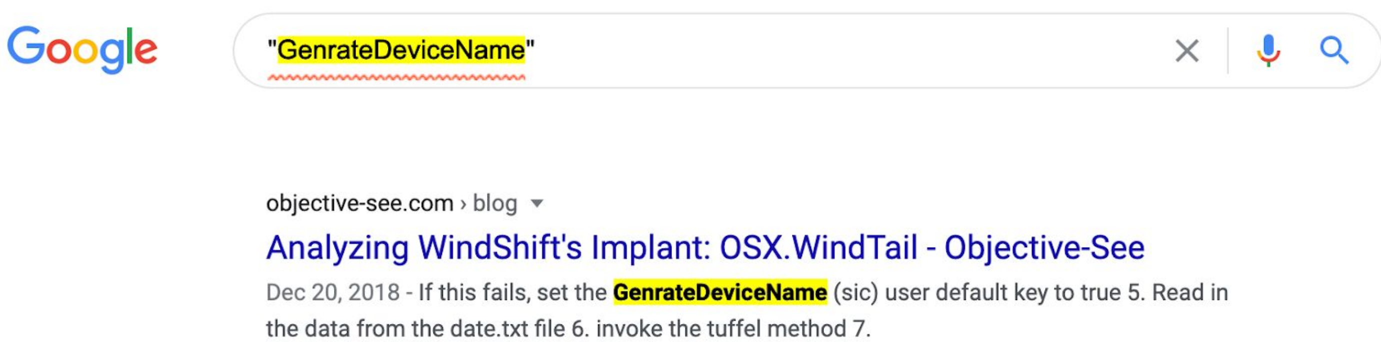
```
$ strings - Final_Presentation.app/Contents/MacOS/usrnode
...

GenrateDeviceName
m_ComputerName_UserName
m_uploadURL

BouCfWujdfbAUfCos/iIOg==
Bk0WPpt0IFFT30CP6ci9jg==
RYfzGQY52uA9SnTjDWCugw==
XCrcQ4M8lnb1sJJo7zuLmQ==
3J1OfDEiMfxgQVZur/neGQ==
Nxv5JOV6nsvg/lfNuk3rWw==
Es1qIvgb4wmPAWwlagmNYQ==

Dop.dat
Fung.dat
song.dat

.zip
/usr/bin/zip
/usr/bin/curl

AES Encryption
```

extracting embedded strings

In the output above, we find:

- Strings that reference survey related logic
- Base-64 encoded strings
- Uniquely named .dat files
- References to macOS utilities (used to compress and upload/download files)
- (AES) encryption

Could this be a backdoor designed to survey and steal files from an infected system? Likely! (Spoiler: it is). And in fact, if we search online for some of the more unique strings (such as the misspelled “GenrateDeviceName” string), we find a match: `OSX.WindTail`:



“GenrateDeviceName” matches OSX.WindTail

> 📝 Note:
>
> Searching online for unique (e.g misspelled) strings can often provide useful results, such as matches to known malware and analysis reports.

Malware authors are of course free to create whatever strings they like. For example, perhaps adding many benign sounding strings in an attempt to mask the true nature of a malicious specimen. Thus, a more comprehensive analysis may be required. However, based on the simplicity of string extractions and the value they can provide, it’s always wise to include it as part of your initial binary triage!

Objective-C Class Information

The majority of Mach-O malware is written in Objective-C. Why is this a good thing for us as malware analysts? Simply put, programs written in Objective-C retain their class declarations when compiled into (Mach-O) binaries. Such class declarations include the name and type of:

- The class
- The class methods
- The class instance variables

In other words, the names (of methods, variables, etc.) that the author used when writing the malware can be extracted from the compiled binary!

Similar to embedded printable strings, this provides (in)valuable insight into many aspects of the malware (such as its capabilities). Insights that can be extracted efficiently, without having to understand any binary code!

> 📝 Note:
>
> As embedded Objective-C class information is (always?) printable strings, this information will (also) show up via the aforementioned strings command.
>
> However, the tools mentioned in this section (i.e. **class-dump**) are designed to specifically extract and reconstruct embedded Objective-C class information, which provides a representation far nearer to the original malware's source code.

There are various utilities designed to extract embedded class information from Mach-O files. A proven favorite is the aptly named class-dump [13] utility (by Steve Nygard).

Here, for example, we use `class-dump` to, extract class information from HackingTeam's persistent Mac backdoor, `OSX.Crisis` [14]:

```
$ class-dump RCSMac.app

...

@interface __m_MCore : NSObject
{
    NSString *mBinaryName;
    NSString *mSpoofedName;
}

- (BOOL)getRootThroughSLI;
- (BOOL)isCrisisHookApp:(id)arg1;
- (BOOL)makeBackdoorResident;
```

```
- (void)renameBackdoorAndRelaunch;
@end
```

(abridged) class-dump output
(OSX.Crisis)

Without having to understand the syntax of Objective-C class declarations, based on instance variable and method names alone, we can ascertain that this binary is malicious and gain insight into its logic. For example, based on the method names "getRootThroughSLI" and "makeBackdoorResident," it is likely that the malware attempts to elevate its privileges to root and persists a backdoor component (perhaps with "spoofed" name)!

📝 Note:

The output from class-dump can also provide valuable input for more involved analysis methods, such as disassembling and/or debugging the binary.

For example, if we're attempting to figure out how OSX.Crisis persists, it would seem prudent to begin analysis at the method named "makeBackdoorResident"!

Another malware specimen that readily spills its secret to `class-dump` is `OSX.Xagent` [15]:

```
$ class-dump Xagent

@interface MainHandler : NSObject
...
- (void)ftpUpload;
- (void)sendKeyLog:(id)arg1;
- (void)stopTakeScreenShot;
- (void)startTakeScreenShot;
- (void)screenShotLoop;
- (void)takeScreenShot;
- (void)deletFileFromPath;
- (void)execFile;
- (void)createFileInSystem;
- (void)downloadFileFromPath;
- (void)readFiles;
- (void)showBackupIosFolder;
- (void)getInstalledAPP;
- (void)remoteShell;
```

```
- (void)getProcessList;
- (void)getInfoOSX;
- (void)getFirefoxPassword;
@end

__attribute__((visibility("hidden")))
@interface InjectApp : NSObject
...
- (void)injectRunningApp;
- (void)sendEventToPid:(id)arg1;
- (BOOL)isInjectable:(id)arg1;
- (id)init;

@end
```

(abridged) class-dump output
(OSX.Xagent)

Based on method names alone, we can extrapolate the malware's (likely) features and capabilities!

📝 Note:

It should be noted that variable and method names of course can be spoofed and/or obfuscated, and thus should be validated via other analysis methods (e.g. a disassembler).

However, such manipulations are a good indication that a binary may be malicious (or at least has something to hide)!

Up Next

In this chapter, we discussed various static analysis tools that can triage unknown Mach-O binaries and assist in their classification. Such tools can often provide enough information to answer the question "is this binary known?" (and as such, already classified as benign or malicious).

However, in the case of a binary appearing to be malicious in nature, yet not matching any known samples, a more comprehensive static analysis tool is needed. This tool is the all powerful disassembler.

In the next chapter, we will introduce some reverse-engineering techniques and discuss how disassemblers (+ decompilers) can be used to fully tear apart any Mach-O binary!

References

1. "OS X ABI Mach-O File Format Reference"
https://github.com/aidansteele/osx-abi-macho-file-format-reference

2. "Parsing Mach-O Files"
https://lowlevelbits.org/parsing-mach-o-files/

3. MachOView
https://sourceforge.net/projects/machoview/

4. "Let's Build A Mach-O Executable"
https://mikeash.com/pyblog/friday-qa-2012-11-30-lets-build-a-mach-o-executable.html

5. "Overview of the Mach-O Executable Format"
https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html

6. WhatsYourSign
https://objective-see.com/products/whatsyoursign.html

7. VirusTotal
https://www.virustotal.com/

8. "Code Signing"
https://developer.apple.com/support/code-signing/

9. "Analyzing OSX/CreativeUpdater"
https://objective-see.com/blog/blog_0x29.html

10. Code Signing – Hashed Out
https://papers.put.as/papers/macosx/2015/CodeSigning-RSA.pdf

11. Technical Note TN2206: macOS Code Signing In Depth
https://developer.apple.com/library/archive/technotes/tn2206/_index.html

12. String's man page
x-man-page://strings

13. Class-Dump
https://github.com/nygard/class-dump

14. “Building HackingTeam's OS X Implant For Fun & Profit”
https://objective-see.com/blog.html#blogEntry6

15. “From Italy With Love?”
https://objective-see.com/blog/blog_0x18.html