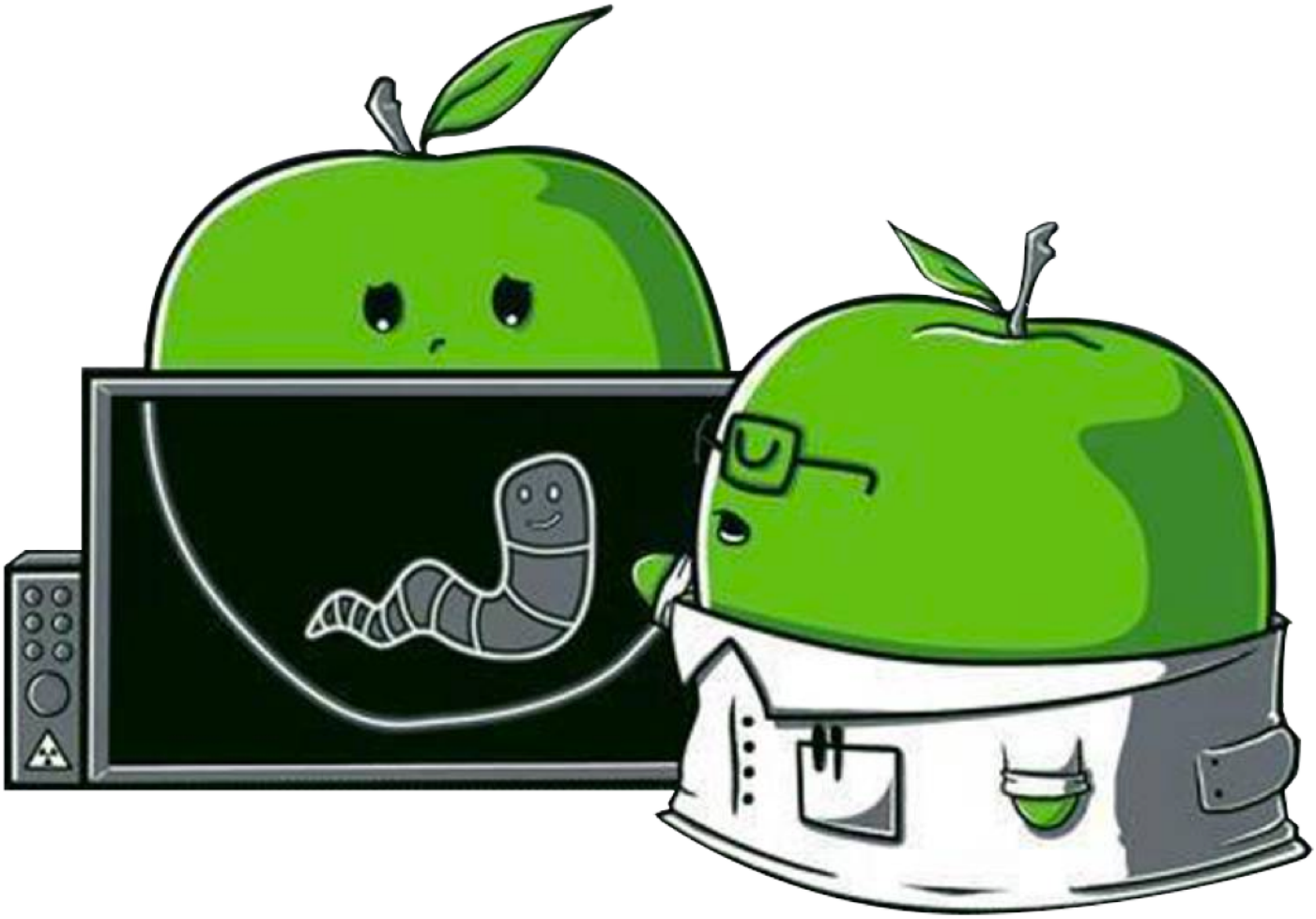


Chapter 0x7: Disassembling & Decompilation

> 📝 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).

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

As the vast majority of Mach-O malware is solely "available" in this compiled binary form (i.e. its source code is not available), we as malware analysts rely on tools that are able to extract meaningful information from such binaries.

In the previous chapter we covered various static analysis tools that can aid in the triage of unknown Mach-O binaries. However, if we truly want to comprehensively understand a Mach-O binary (for example a specimen that appears to be a new piece of Mac malware), other more sophisticated tools are required.

Advanced reverse-engineering tools offer the ability to disassemble, decompile (and even dynamically debug) binaries. In this chapter, we'll stick to the static analysis approaches of disassembling and decompilation (though in later chapters we'll cover dynamic debugging as well). While these tools require at least an elementary understanding of low-level reversing concepts (such as assembly code), and may lead to time-consuming analysis sessions, their analysis abilities are invaluable and unmatched. Even the most sophisticated malware specimen is no match for a skilled analyst wielding these tools!

Before discussing the specifics of disassemblers and decompilers, a brief foray into assembly code is required.

Assembly Language Basics

> 📝 Note:
>
> Entire books have been written on the topics of disassembling binary code and the assembly language.
>
> Here, we provide only the basics (and take some liberties in simplifying various concepts), and assume the reader is familiar with various basic reversing concepts (such as registers, etc.).
>
> Two excellent books on the topic of reverse-engineering (including assembly/disassembly) are:
>
> - "Hacker Disassembling Uncovered" [1]
> - "Reversing: Secrets of Reverse Engineering" [2]

Software (including malware) is written in a programming language ...an unambiguous "human friendly"-ish language that may then be translated (compiled) into binary code. Scripts that we discussed in Chapter 0x5 ("Non-Binary Analysis"), are not compiled per se, but rather "interpreted" at runtime into commands or code that the system understands.

As noted, when analyzing a compiled Mach-O binary suspected of being malicious, the original source code is generally not available. We must leverage a tool that can understand the compiled binary machine-level code, and translate it back into something more readable: assembly code! This process is known as disassembling.

Assembly is a low-level programming language that is translated directly to binary instructions. This direct translation means that binary code within a compiled binary can (later) be directly compiled back into assembly. For example, the binary sequence: `10010001000001111000000000111000` can be represented in assembly code as: `add rax, 0x38` ("*add 38 hex to the* `rax` *register*").

At its core, a disassembler takes as input a compiled binary (such as a malware sample) and performs this translation back into assembly code. Of course, it's up to us to make sense of the provided assembly!

> 📝 Note:
>
> There are various "versions" of assembly. We'll focus on x86_64 (the 64-bit version of the x86 instruction set), the System V ABI (calling convention) with Intel syntax.
>
> ...as this is the (current) instruction set and calling convention of macOS!

Assembly instructions are "*represented by a mnemonic which [is], often combined with one or more operands*" [3]. Mnemonics generally describe the instruction:

Mnemonic	**Example**	**Description**
`add`	`add rax, 0x100`	Adds the second operand (e.g. 0x100) to the first.
`mov`	`mov rax, 0x100`	Moves the second operand (e.g. 0x100) into the first.
`jmp`	`jmp 0x100000100`	Jump to (i.e. continue execution at) the address in the operand.
`call`	`call rax`	Execute the subroutine specified at the address in the operand.

Generally, operands are either registers (a named memory 'slot' on the CPU) or numeric values. Some of the registers you'll encounter while reversing a 64-bit Mach-O binary

include, rax, rbx, rcx, rdx, rdi, rsi, rbp, rsp, and r8 - r15. As we'll see shortly, oftentimes specific registers are consistently used for specific purposes, which simplifies reverse-engineering efforts.

> 📝 Note:
>
> All 64-bit registers can also be "referenced" by their 32-bit (or smaller) components ...which you'll (still) come across during binary analysis.
>
> *"All registers can be accessed in 16-bit and 32-bit modes. In 16-bit mode, the register is identified by its two-letter abbreviation from the list above. In 32-bit mode, this two-letter abbreviation is prefixed with an 'E' (extended). For example, 'EAX' is the accumulator register as a 32-bit value.*
>
> *Similarly, in the 64-bit version, the 'E' is replaced with an 'R' (register), so the 64-bit version of 'EAX' is called 'RAX'."* [3]

Before we wrap up our (cursory) discussion of assembly code, let's briefly discuss calling conventions. This will give us an understanding of how API (method) calls are made, how arguments are passed in, and how the response is handled ...in assembly code.

Why is this relevant? Well, one can often gain a fairly comprehensive understanding of a Mach-O binary by simply studying the system API methods it invokes. For example, a malicious binary that makes a call to a "write file" API method, passing in both a property list and path that falls within the ~/Library/LaunchAgents directory, is likely persisting as a launch agent!

Thus, we often don't need to spend hours understanding all assembly instructions in a binary, but instead can focus on the instructions "around" API calls to understand:

- What (API) calls are invoked
- What arguments are passed in to the (API) call
- What actions it takes based on the result of the (API) call

...often this understanding is sufficient to gain a relatively comprehensive understanding of the (potentially malicious) binary specimen we're analyzing.

To facilitate the explanation of calling conventions and method calls (at the assembly level), we'll focus on a snippet Objective-C, which creates a NSURL object that is initialized with "www.google.com":

```
01    //url object
02    NSURL* url = [NSURL URLWithString:@"www.google.com"];
```

When a program wants to invoke a method or a system API call, it first needs to "prepare"

the arguments for the call. In the source code above, when invoking the `URLWithString:` method (which expects a string object as its only argument), the Objective-C code passes in the string "www.google.com".

At the assembly level, there are specific “rules” about how to pass arguments to a method or API function. This is referred to as the calling convention. The rules of the calling convention are articulated in an Application Binary Interface (ABI), and for 64bit macOS system are as follows:

Argument	**Register**
1st argument	`rdi`
2nd argument	`rsi`
3rd argument	`rdx`
4th argument	`rcx`
5th argument	`r8`
6th argument	`r9`

macOS (intel 64bit) calling convention

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

Thus, once a call is identified in the disassembly (by the `call` mnemonic), looking backwards in the assembly code will reveal the values of the arguments passed to the method or API. This can often provide valuable insight into the code’s logic (i.e. what URL a malware sample is attempting to connect to, the path of a file it’s opening, etc.).

And what about when the `call` instruction returns? Consulting the ABI reveals that the return value of the method or API call will always be stored in the `rax` register. Thus once the `NSURL`’s `URLWithString:` method call returns, the newly constructed NSURL object will be in the `rax` register.

As the `rax` register holds the return value when the call instruction completes, you’ll often see disassembly with a `call` instruction, immediately followed by instructions checking and taking an action based on the result of the value in the `rax` register. For example (as we’ll see shortly) a malicious sample choosing not to infect a system if a function that checks for network connectivity returns zero (NO/false) in the `rax` register.

Something else that is imperative to understand when reversing Objective-C binary code is

the `objc_msgSend` [4] function.

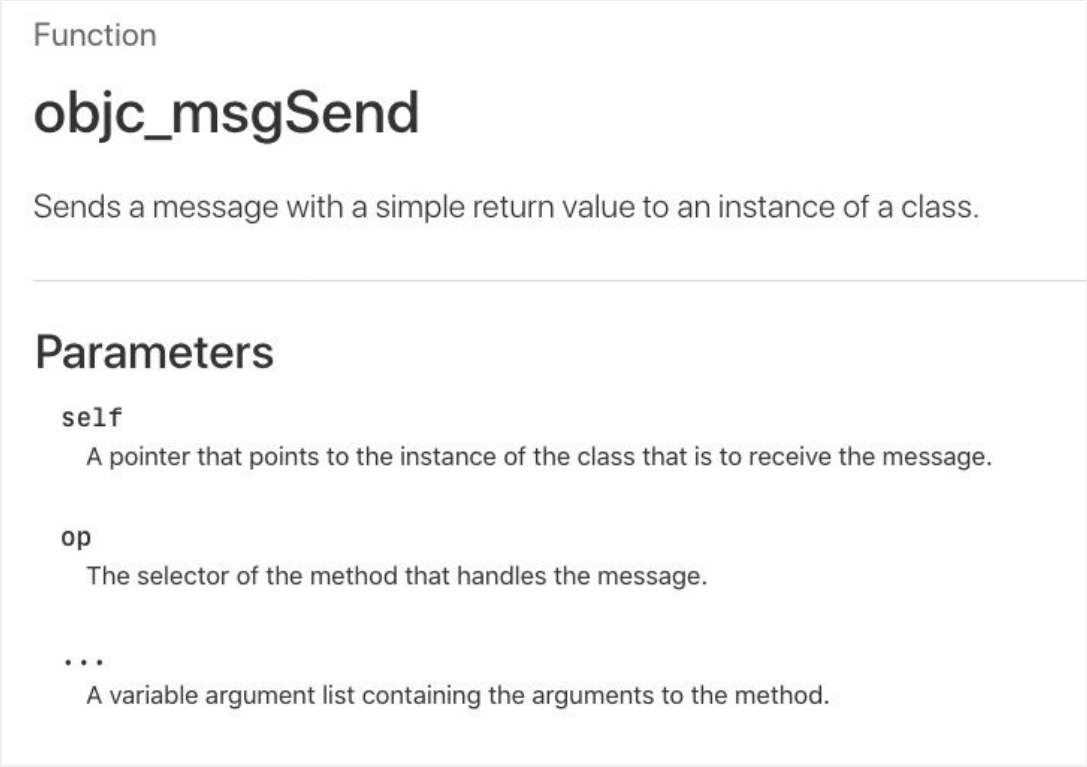
Recall the following Objective-C code that simply constructs a URL object:

```
01    //url object
02    NSURL* url = [NSURL URLWithString:@"www.google.com"];
```

When this code is compiled, the compiler (`llvm`) will translate this Objective-C call (and most other Objective-C calls), into code that invokes the `objc_msgSend`. Or as Apple explains:

> *"When it encounters a [Objective-C] method call, the compiler generates a call to ...objc_msgSend"* [4]

Apple developer documentation contains an entry for this function, stating that it "*sends a message with a simple return value to an instance of a class*" [4]:



The objc_msgSend function

As the vast majority of Objective-C calls are routed through this function, it is imperative to understand it when reversing compiled Objective-C code. So, let's break it down!

First, what does "*sends a message ...to an instance of a class*" even mean? Simply put, this means invoking (calling) an object's method.

> 📝 Note:
>
> The Objective-C runtime is based on the notion of sending messages, and other rather unique object originated paradigms.
>
> For an in-depth discussion of the Objective-C runtime and its internals, consult the following by nemo:
>
> - "Modern Objective-C Exploitation Techniques" [5]
> - "The Objective-C Runtime: Understanding and Abusing" [6]

And second, what about `objc_msgSend`'s parameters:

- The first parameter (`self`) is "*a pointer that points to the instance of the class that is to receive the message*" [4]. Or more simply put, it's the object that the method is being invoked upon. If the method is a class method, this will be an instance of the class object (as a whole), whereas for an instance method, `self` will point to an instantiated instance of the class as an object.
- The second parameter, (`op`), is "*the selector of the method that handles the message*" [4]. Again, more simply put, this is just the name of the method.
- The remaining parameters are any values that are required by the method (`op`).

Finally, `objc_msgSend` returns whatever the method (`op`) returns.

Recall that the ABI defines how arguments are passed to a function call. As such, we can map exactly which registers will hold `objc_msgSend`'s arguments at time of invocation:

Argument	**Register**	**(for) `objc_msgSend`**
1st argument	`rdi`	`self`: object that the method is being invoked upon
2nd argument	`rsi`	`op`: name of the method
3rd argument	`rdx`	1st argument to the method
4th argument	`rcx`	2nd argument to the method

5th argument	r8	3rd argument to the method
6th argument	r9	4th argument to the method
7th+ argument	rsp+ (on the stack)	5th+ argument to the method

Of course the registers rdx, rcx, r8, r9, are only used if the method being invoked requires them (for arguments). For example, a method that only takes one argument will only utilize the rdx register.

Also, like any other function or method call, once the call to objc_msgSend completes, the rax register will hold the return value (which is actually the return value from the method that was invoked).

This wraps up our very brief discussion on assembly language basics. Armed with foundation understanding of this low-level language, let's now look at disassembling binary code.

Disassembling

Disassembling involves converting binary code (1s and 0s) back into assembly instructions. This assembly code can then be analyzed to gain a comprehensive understanding of the binary. A disassembler (discussed shortly) is a program that is able to perform this translation and facilitate the analysis of compiled binaries.

Here, we'll discuss various disassembling concepts, illustrated via real world examples (taken directly from malicious code). It is important to remember that generally speaking, the goal of analyzing a malicious code is to gain a comprehensive understanding of its logic and capabilities ...not necessarily to understand each and every assembly instruction. As we noted earlier, focusing on the logic around method and function calls can often provide an efficient means to gain such an understanding.

As such, let's briefly look at an example of disassembled code in order to illustrate how to identify such calls, the parameters, and the (API) response. The end result? A comprehensive understanding of the disassembled code snippet.

Malware sometimes contains logic to check if its host is connected to the internet. If the infected system is offline, the malware will often wait (sleep) before trying to connect to its command and control server for tasking.

A specific example of malware that checks for network connectivity is OSX.Komplex [7], which contains a function named connectedToInternet. By studying the disassembled binary code of this nation-state backdoor, we can confirm this function indeed checks if the infected system is online as well as understand how it accomplishes this check.

Specifically our analysis will reveal the malware checks for network connectivity via Apple's `NSData` class, invoking the `dataWithContentsOfURL:` method [8]. If a remote URL (www.google.com) is not reachable (i.e. the infected system is offline), the call will fail, indicating the system is offline.

Now let's dive into the disassembly of `OSX.Komplex`'s `connectedToInternet` function (annotated for clarity). Note that we'll break down the function piece by piece and first show an Objective-C representation, reconstructed from the disassembly.

```
01  connectedToInternet() {
02
03    //url object
04    NSURL* url = [NSURL URLWithString:@"www.google.com"];
```

Previously we mentioned that Objective-C methods calls are "translated" into calls to the the `objc_msgSend` function. Thus, it's unsurprising to see a call to this function in the disassembly:

```
01  connectedToInternet
02
03  ;move a pointer to the NSURL class into rdi
04  ; the rdi register holds the first parameter ('self')
05  mov     rdi, qword [objc_cls_ref_NSURL]
06
07  ;move a pointer to the method name 'URLWithString:' into rsi
08  ; the rsi register holds the 2nd parameter ('op')
09  lea     rsi, qword [URLWithString:]
10
11  ;load the address of the url in rdx
12  ; the rdx register holds the 3rd parameter, which is the 1st parameter passed to
13  ; the method being invoked (URLWithString:)
14  lea     rdx, qword [_www_google_com]
15
16  ;move a pointer to objc_msgSend into the rax register
17  ; and then invoke it
18  mov     rax, cs:_objc_msgSend_ptr
19  call    rax
20
21  ;save the response into a (stack) variable named 'url'
    ; the rax register holds the result of the method call
    mov         qword [rbp+url], rax
```

We also see that the single line of Objective-C code, (`NSURL* url = [NSURL URLWithString:@"www.google.com"],`) was translated into several lines of assembly code.

First the parameters are initialized (in the expected registers) for a call to `objc_msgSend`, the call is then made, and the result is saved.

Specifically, the `rdi` register (the first parameter) is loaded with a reference to the `NSURL` class. Then, the second parameter (`rsi`) is loaded with the name of the method: `URLWithString:`. Finally `rdx` is initialized with the string "www.google.com". Now the `objc_msgSend` can be made. Once the call completes, the newly initialized `NSURL` object is returned in the `rax` register and stored into a local variable.

Once a `NSURL` object has been constructed the malware invokes the `NSData`'s `dataWithContentsOfURL:` method. Again, before looking at the disassembly, let's construct a likely representation in Objective-C:

```
01    //data object
02    // initialized by trying to connect/read to google.com
03    NSData* data = [NSData dataWithContentsOfURL:url];
```

Here's the (relevant) disassembly code of `OSX.Komplex`'s `connectedToInternet` method:

```
01  ;the following code prepares the relevant registers
02  ; then makes an objective-c call via the objc_msgSend function
03
04  ;move a pointer to the NSData class into rdi
05  ; the rdi register holds the first parameter ('self')
06  mov     rdi, qword [objc_cls_ref_NSData]
07
08  ;move a pointer to the method name 'dataWithContentsOfURL:' into rsi
09  ; the rsi register holds the 2nd parameter ('op')
10  lea     rsi, qword [dataWithContentsOfURL:]
11
12  ;mov the (previously created) url object into rdx
13  ; the rdx register holds the 3rd parameter, which is the 1st parameter passed to
14  ; the method being invoked ('dataWithContentsOfURL:')
15  mov     rdx, qword [rbp+url]
16
17  ;move a pointer to objc_msgSend into the rax register
18  ; and then invoke this function, to make the objective-c call
19  mov     rax, cs:_objc_msgSend_ptr
```

```
20  call    rax
21
22  ;save the response into a (stack) variable named data
23  ; the rax register holds the result of the method call
24  mov        qword [rbp+data], rax
```

Similar to the disassembly for the call into `NSURL`'s `URLWithString:` method, here we see the parameters being initialized (in the expected registers) for a call to `objc_msgSend`, the call is then made, and the result is saved (into a variable named `data`).

`OSX.Komplex`'s `connectedToInternet` function completes by returning an integer value (0x0/0x01) to the caller, based on the result of `NSData`'s `dataWithContentsOfURL:` method. Specifically, a 0x1 ('true') is returned if the method succeeded to indicate the malware was able to connect to the internet and reach google.com. If the `dataWithContentsOfURL` method failed (meaning it returned a blank (nil) data object), the `connectedToInternet` function returns 0x0 ('false') to indicate to the caller that the network is unreachable.

The malware authors likely wrote something similar to the following Objective-C code to implement this return-value logic:

```
01    //set flag
02    // YES (true) if google was reachable
03    isConnected = (data != nil) ? YES : NO;
04    return (int)isConnected;
```

And how does this look like in (disassembled) assembly code? Glad you asked:

```
01  ;compare the the data variable with zero (nil)
02  cmp    qword [rbp+data], 0x0
03
04  ;if data was zero,
05  ; jump to the 'notConnected' label
06  je     notConnected
07
08  ;set 'isConnected' to 0x1
09  mov byte [rbp+isConnected], 0x1
10
11  ;skip over the 'notConnected' logic
12  jmp    leave
13
14 notConnected:
```

```
15
16   ;set 'isConnected' to 0x0
17   mov    byte [rbp+isConnected], 0x0
18
19  leave:
20
21   ;move the value into rax
22   ; note: al is the lower byte of rax
23   mov     al, byte [rbp+isConnected]
24   and     al, 0x1
25   movzx   eax, al
26
27   return
```

First the `cmp` instruction is used to compare the value of the `data` variable (returned from the call to `dataWithContentsOfURL`). If it's 0 (nil), the assembly code jumps to the `notConnected` label and sets the value of the `isConnected` variable to 0. Otherwise, if the `dataWithContentsOfURL` method returned a non-nil value, the `isConnected` variable is set to one.

Finally, the `isConnected` variable is moved into the `rax` (`eax`) register by means of a few instructions. Such instructions are required to ensure the boolean value is correctly converted into a (larger) integer value to be returned to the caller.

As is often the case, a few lines of Objective-C code are often expanded into many assembly instructions, which makes analyzing disassembled code rather time consuming. However without access to source code, often we have little other choice. And, the assembly instructions do provide unparalleled insight into the malware's inner workings ...so much so that often we can completely reconstruct the malware's code in a higher-level language. Here for example a complete reconstruction of the `connectedToInternet` function:

```
01  int connectedToInternet()
02  {
03     //result
04     BOOL isConnected = NO;
05
06     //url object
07     // let's use google.com
08     NSURL* url = [NSURL URLWithString:@"www.google.com"];
09
```

```
10
11      //data object
12      // init'd by trying to connect/read to google.com
13      NSData* data = [NSData dataWithContentsOfURL:url];
14
15      //set flag
16      // YES (true) if google was reachable!
17      isConnected = (data != nil) ? YES : NO;
18
19      return (int)isConnected;
    }
```

reconstruction of a connectivity check (OSX.Komplex)

Now, let's walk through the (annotated) disassembly of malware's code that both invokes the `connectedToInternet` function, and then acts upon its response.

```
01  isConnected:
02
03  ;call the function
04  call        connectedToInternet()
05
06  ;check a 0x0 or 0x1 was returned
07  and         al, 0x1
08  mov         byte [rbp+isConnected], al
09  test        byte [rbp+isConnected], 0x1
10
11  ;take this if 0x0 (not connected)
12  jz          notConnected
13
14  ;take this if 0x1 (connected)
15  jmp         continue
16
17  ;sleep
18  notConnected:
19  mov         edi, 0x3c
20  call        sleep
21
22  ;check connection (again)
23  jmp         isConnected
24
```

```
25  continue:
26  ...
```

invoking connectedToInternet, and processing the result (OSX.Komplex)

First the code invokes the connectedToInternet function. As this function takes no parameters, no register setup is required. Following the call the malware checks if the return value is 0x0 (NO/false). This is accomplished via a test and a jz (jump zero) instruction. The test instruction *"performs a bitwise AND on two operands"* [9] and sets the zero flag based on the result. Thus if the connectedToInternet function returns a zero, the jz instruction will be taken, jumping to the notConnected label. Here, the code invokes the sleep function ...before jumping back to the isConnected label, to check for connectivity once again. In other words, the malware will wait until the system is connected to the internet, before continuing on.

With this comprehensive understanding, we can (re)construct this logic in the following Objective-C code:

```
01  while(0x0 == connectedToInternet()) {
02              sleep(0x3c);
03  }
```

...in Objective-C

Of course not all Mac binaries (including malware) are written in Objective-C. Let's look at another (abridged and annotated) snippet of disassembly - this time from a Lazarus Group first-stage implant loader (originally written in C++) [10]. Specifically, we'll walk through a snippet of assembly code from a function named getDeviceSerial:

```
01  ;function: getDeviceSerial(char*)
02  ; first arg (rdi): output buffer ...for device serial #
03  ; return (rax): status (success/error)
04
05   ;move pointer to output buffer into r14
06   mov      r14, rdi
07
08   ;move kIOMasterPortDefault into r15 register
09   mov     rax, qword [_kIOMasterPortDefault]
10   mov     r15d, dword [rax]
11
12
```

```
13
14  ;invoke IOServiceMatching
15  ;1st arg (rdi): the string "IOPlatformExpertDevice"
16  lea     rdi, qword [IOPlatformExpertDevice]
17  call    IOServiceMatching
18
19  ;invoke IOServiceGetMatchingService
20  ; 1st arg (rdi): kIOMasterPortDefault
21  ; 2nd arg (rsi): result of the call to IOServiceMatching
22  mov     edi, r15d
23  mov     rsi, rax
24  call    IOServiceGetMatchingService
25
26  ;invoke IORegistryEntryCreateCFProperty
27  ; 1st arg (rdi): result of the call to IOServiceGetMatchingService
28  ; 2nd arg (rsi): the string "IOPlatformSerialNumber"
29  ; 3rd arg (rdx): the (default) allocator kCFAllocatorDefault
30  ; 4th arg (rcx): the options
31  mov     r15d, eax
32  mov     rax, qword [_kCFAllocatorDefault]
33  mov     rdx, qword [rax]
34  lea     rsi, qword [IOPlatformSerialNumber]
35  xor     ecx, ecx
36  mov     edi, r15d
37  call    IORegistryEntryCreateCFProperty
38
39  ;invoke CFStringGetCString
40  ; 1st arg (rdi): result of the call to IORegistryEntryCreateCFProperty
41  ; 2nd arg (rsi): the output buffer
42  ; 3rd arg (rdx): the buffer size
43  ; 4th arg (rcx): the encoding
44  mov         edx, 0x20
45  mov         ecx, 0x8000100
46  mov         rdi, rax
47  mov         rsi, r14
48  call        CFStringGetCString

    return
```

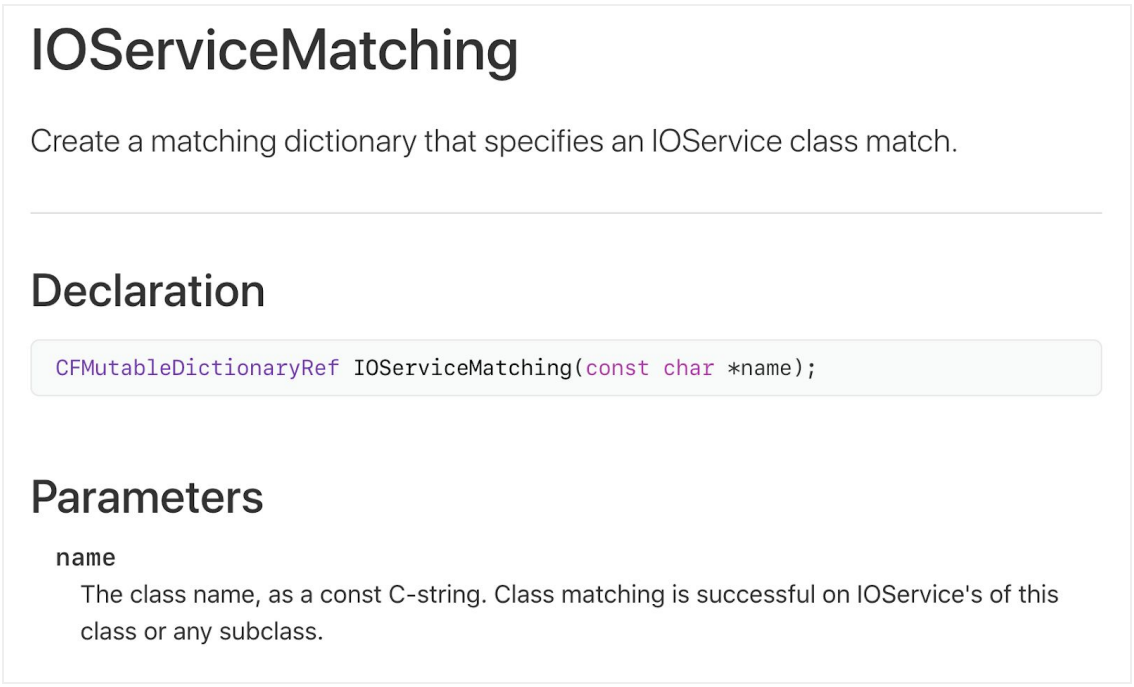
...definitely a more sizable chunk of assembly code! But not to worry, we'll walk through it in detail.

First, observe that the disassembler has extracted function declaration, which (luckily for us) includes its original name as well as the number and format of its parameters. From the name, `getDeviceSerial`, let's assume (though we'll also validate) that this function will retrieve the serial number of the infected system. Since the function takes as its only parameter, a pointer to a string buffer (`char*`), it seems reasonable to assume the function will store the extracted serial number in this buffer (so that it is available to the caller).

Starting at line #06, we see the function first moves this argument (recall `rdi` always holds the 1st argument), the address of the output buffer, into the `r14` register. Why? As noted, the `rdi` register is initialized with the first argument for any function call. If the `getDeviceSerial` function makes any *other* calls (which it does), the `rdi` register will have to be reinitialized (for those other calls). Thus, the function must 'save' the address of the output buffer into another (non-used) register, so that this address may be used later ...for example, at the end of the function when it's populated with the extracted serial number.

The function then (lines #09 - 10) moves a pointer to `kIOMasterPortDefault` into `rax`, and dereferences it into the `r15` register. According to Apple developer documentation, the `kIOMasterPortDefault` is "*The default mach port used to initiate communication with IOKit.*" [11] Seems likely the malware will be communicating with IOKit as the means to extract the infected device's serial number.

In lines 14 and 15, the function `getDeviceSerial` makes its first call into an Apple API: the `IOServiceMatching` function. Apple notes this function creates "*a matching dictionary that specifies an IOService class match*" taking in a single parameter, and returning the matching dictionary [12]:



IOServiceMatching

Create a matching dictionary that specifies an IOService class match.

Declaration

```
CFMutableDictionaryRef IOServiceMatching(const char *name);
```

Parameters

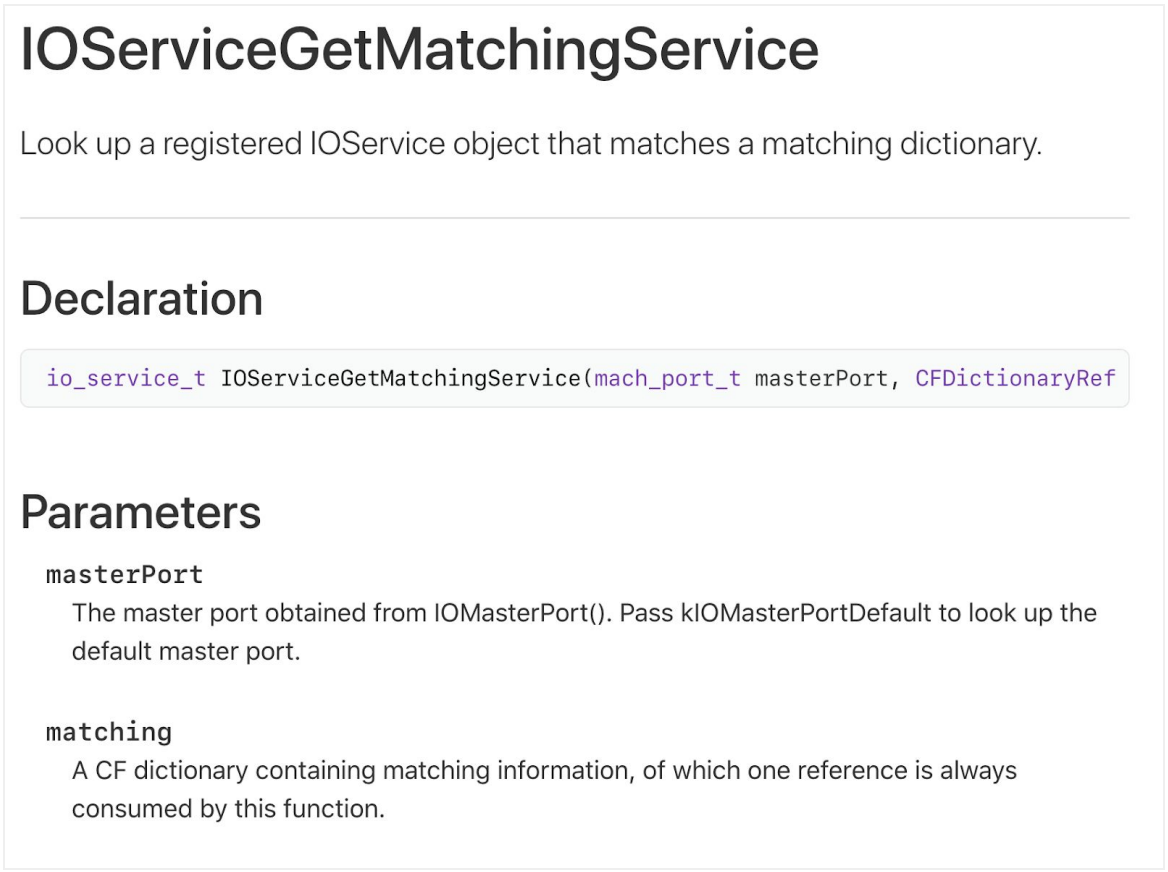
name

The class name, as a const C-string. Class matching is successful on IOService's of this class or any subclass.

the `IOServiceMatching` function

We know that when making a call to a function or method, the rdi register holds the first argument. In line #14, we see the assembly code initialize this register with the value of "`IOPlatformExpertDevice`". In other words, it's invoking the `IOServiceMatching` function with the string "`IOPlatformExpertDevice`".

Once the matching dictionary has been created, the code invokes the `IOServiceGetMatchingService` function (line # 22). Apple documents state that this function will "*look up a registered `IOService` object that matches a matching dictionary.*" [14]. For parameters, it expects a master port and a matching dictionary:



IOServiceGetMatchingService

Look up a registered IOService object that matches a matching dictionary.

Declaration

```
io_service_t IOServiceGetMatchingService(mach_port_t masterPort, CFDictionaryRef
```

Parameters

masterPort
The master port obtained from IOMasterPort(). Pass kIOMasterPortDefault to look up the default master port.

matching
A CF dictionary containing matching information, of which one reference is always consumed by this function.

the `IOServiceGetMatchingService` function

On line #20, the assembly code moves a value from the `r15` register into the `edi` register (the 32bit part of the `rdi` register). Looking back to line numbers 9-10, we see the code previously moving the `kIOMasterPortDefault` into the `r15` register. The code on line #20 is simply moving `kIOMasterPortDefault` into the `edi` register (as the first argument for the call to `IOServiceGetMatchingService`).

On line #21, we see `rax` being moved into the `rsi` register (recall the `rsi` register is used as the 2nd parameter for function calls). And (following a function call), the `rax` register holds the result of the call. This means the `rsi` register will contain the matching dictionary from the call to `IOServiceMatching` (made on line #15).

After the call to `IOServiceGetMatchingService`, an `io_service_t` service is returned (in the rax register). Specifically, a service that matches `IOPlatformExpertDevice`.

Next, the code sets up the parameters for a call to the `IORegistryEntryCreateCFProperty` function, which Apple documentation states *"creates an instantaneous snapshot of a registry entry property."* [14] In other words, the code is extracting the value of some (IOKit) registry property. But which one?

IORegistryEntryCreateCFProperty

Create a CF representation of a registry entry's property.

Declaration

```
CFTypeRef IORegistryEntryCreateCFProperty(io_registry_entry_t entry, CFString
Ref key, CFAllocatorRef allocator, IOOptionBits options);
```

Parameters

entry
The registry entry handle whose property to copy.

key
A CFString specifying the property name.

allocator
The CF allocator to use when creating the CF container.

options
No options are currently defined.

The parameter setup for the call to the `IORegistryEntryCreateCFProperty` function begins by loading the kCFAllocatorDefault into the `rdx` register (lines #29-13). The `rdx` register is used for the 3rd argument, which for the call to `IORegistryEntryCreateCFProperty` is the *"allocator to use"* [12].

Next (line #32), the address of the string "`IOPlatformSerialNumber`" is loaded into the `rsi` register. As the `rsi` register is used for the 2nd argument, this (according to Apple's documentation for the `IORegistryEntryCreateCFProperty` function) is the property name of interest!

On line #33, `rcx`, the 4th argument ("options"), is initialized to zero (xoring of oneself, sets oneself to zero). Finally, before making the call, the value from `r15d` is moved into the 32bit part of the `rdi` register (`edi`). This has the effect of initializing

the first parameter (rdi) with the value of kIOMasterPortDefault (previously stored in r15d).

After the call to IORegistryEntryCreateCFProperty, the rax register will hold the value of the required property: IOPlatformSerialNumber.

Finally, the function invokes the CFStringGetCString function to convert the extracted property (which is (CF)string object) to a plain null-terminated "C-string". Of course, the parameters have to be initialized prior to this call (lines #42-45).

The edx register (the 32bit part of the rdx, argument #3) is set to 0x20, which specifies the output buffer size. Then the ecx register (the 32bit part of the rcx, argument #4) is set to the kCFStringEncodingUTF8 (0x8000100). The first argument (rdi) is set to the value of rax, which is the result of the call to IORegistryEntryCreateCFProperty: the extracted property value of IOPlatformSerialNumber.

Finally, the 2nd argument (rsi) is set to r14. And what is in the r14 register? Scrolling back all the way to line #6, we see it comes from rdi, which is (was) the value of the parameter passed to the getDeviceSerial. Since Apple's documentation for CFStringGetCString states the 2nd argument is the *"C string buffer into which to copy the string,"* [15] we now know the parameter passed to the getDeviceSerial function is a buffer for a string!

This completes our (very thorough!) analysis of the malware's getDeviceSerial function. By focusing on the API calls made by this function, we were able to ascertain its exact functionality: the retrieval of the infected system's serial number (IOPlatformSerialNumber) via IOKit. Moreover, via parameter analysis, we were able to determine that the getDeviceSerial function would be invoked with a buffer for the serial number.

...who needs source code right!?

However at this point, we can all agree that reading assembly code is rather tedious. Luckily, due to recent advances in decompilers, there is hope!

Decompilation

Given a binary, such as a Mach-O, a disassembler can parse the file and translate the binary code back into human-readable assembly, thus allowing detailed analysis to commence.

Decompilers seek to take this translation one step further by recreating a source-code level representation of extracted binary code. Source-code (i.e. C or Objective-C) representation is both more succinct and "readable" than (dis)assembly, making analysis of unknown binaries a simpler task.

Recall the `getDeviceSerial` function from the Lazarus Group first-stage implant loader. The full disassembly of this function is about 50 lines. The decompilation? ...around 15:

```
01  int getDeviceSerial(int * arg0) {
02      r14 = arg0;
03      ...
04      r15 = kIOMasterPortDefault;
05      rax = IOServiceMatching("IOPlatformExpertDevice");
06      rax = IOServiceGetMatchingService(r15, rax);
07      if (rax != 0x0) {
08          rbx = CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
09                @"IOPlatformSerialNumber", kCFAllocatorDefault, 0x0), r14, 0x20,
10                kCFStringEncodingUTF8) != 0x0 ? 0x1 : 0x0;
11          IOObjectRelease(rax);
12      }
13      rax = rbx;
14      return rax;
15  }
```

`getDeviceSerial` decompiled

The decompilation is quite readable, and thus it is relatively easy to understand the logic of this function!

Similarly, the `connectedToInternet` function discussed early in the chapter, decompiles decently as well (though the decompiler does see a little confused by the Objective-C syntax ...though, who isn't?):

```
01  int connectedToInternet()
02  {
03      if( (@class(NSData), &@selector(dataWithContentsOfURL:), (@class(NSURL),
04           &@selector(URLWithString:), @"http://www.google.com")) != 0x0)
05      {
06           var_1 = 0x1;
07      }
08      else {
```

```
09          var_1 = 0x0;
10      }
11      rax = var_1 & 0x1 & 0xff;
12      return rax;
13  }
```

`connectedToInternet` *decompiled (OSX.Komplex)*

> 📝 Note:
>
> Taking into consideration the many benefits of decompilation over disassembly, one may be wondering why disassembling was discussed at all.
>
> First, even the best decompilers occasionally struggle to analyze complex binary code (such as malware with anti-analysis logic). Disassemblers that simply translate binary code (vs. attempt to (re)create source-code level representations) are far less susceptible. Thus, "dropping down" to the assembly level code provided by the disassembler may be the only option.
>
> Second, as we saw in the above decompilation of the getDeviceSerial and connectedToInternet functions, assembly code concepts (such as registers) are still present in the code, and thus relevant.
>
> While decompilation can greatly simplify the analysis of binary code, the ability to understand (dis)assembly code is arguably a foundational skill in comprehensive malware analysis.

Hands on With Hopper

So far, we've discussed the concepts of disassembly and decompilation without mentioning specific tools which provide these services. Such tools can be somewhat complex and thus a bit daunting to the beginner malware analyst. As such, here we'll briefly discuss one such tool (Hopper), providing a high-level, hands-on "quick start" guide to binary analysis!

Hopper [16] is described by its creators as a,

> *"reverse engineering tool that lets you disassemble, decompile and debug your applications."* [16]

Reasonably priced and designed natively for macOS, Hopper boasts a powerful disassembler and decompiler that excels at analyzing Mach-O binaries. It's a solid choice for Mac malware analysis.

> 📝 Note:
>
> A free demo version of Hopper is available from:
>
> https://www.hopperapp.com/download.html
>
> If you're familiar with or fond of another (perhaps more powerful) disassembler / decompiler (such as IDA Pro or Ghidra), the specifics of this section may not apply. However, at a conceptual level, they are broadly applicable across most reverse-engineering tools.

In this brief introduction to Hopper, we'll disassemble and decompile Apple's standard "Hello World" (Objective-c) code:

```
01  #import <Foundation/Foundation.h>
02
03  int main(int argc, const char * argv[]) {
04      @autoreleasepool {
05          // insert code here...
06          NSLog(@"Hello, World!");
07      }
08      return 0;
09  }
```

Apple's "Hello World" template code

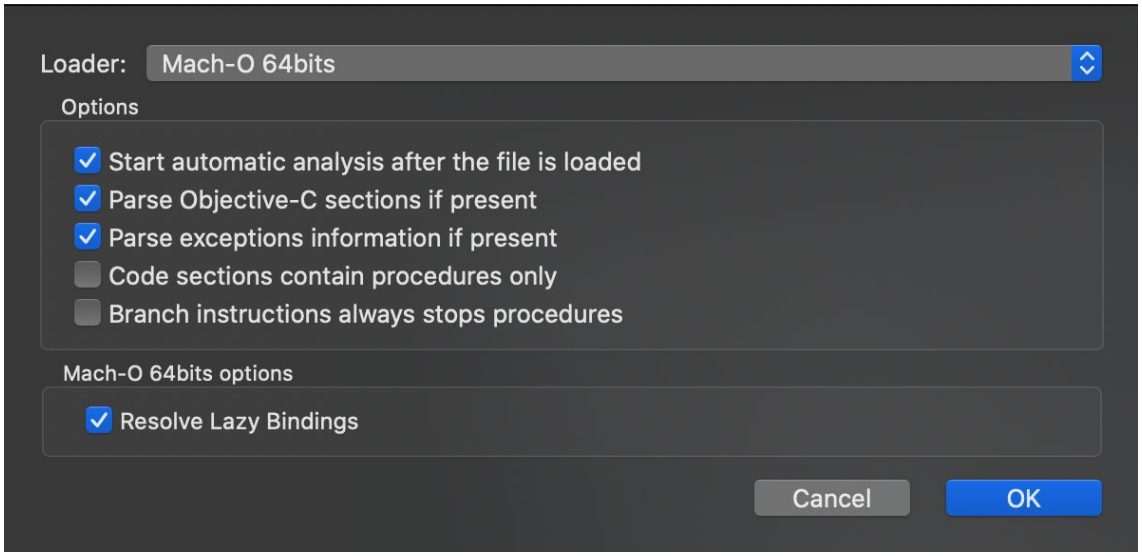
Though trivial it affords us with an example binary, sufficient for illustrating many of Hopper's features and capabilities. An understanding of such features and capabilities, of course, is imperative for the analysis of more complex (malicious) binaries.

We start by compiling the above Objective-C code, and confirm it is now (as expected), a standard 64-bit Mach-O binary:

```
$ file helloWorld/Build/Products/Debug/helloWorld

helloWorld: Mach-O 64-bit executable x86_64
```

First, launch Hopper.app. To start analysis of our `helloWorld` (or any) Mach-O binary simply choose: `File -> Open (⌘+O)`. Select the Mach-O binary for analysis and in the loader window that is shown leave the defaults selected, and click 'OK':



Loader Window
(Hopper.app)

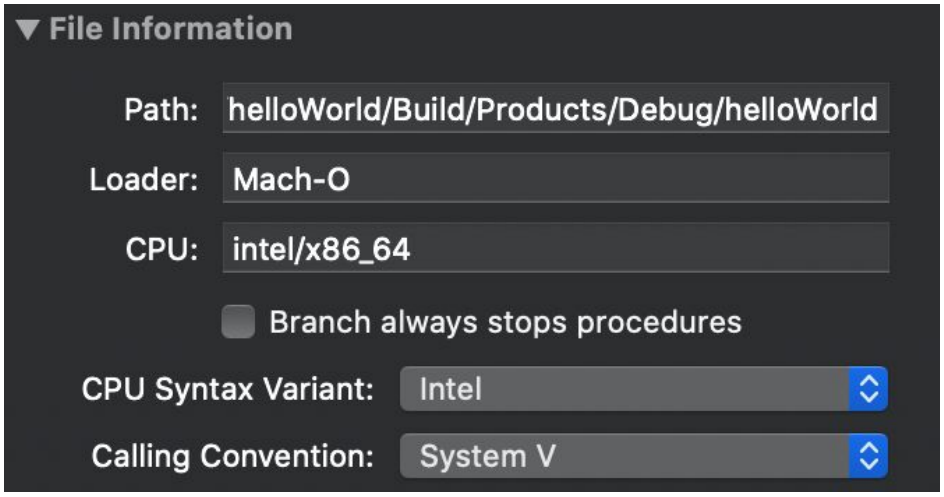
Hopper will automatically begin analysis of the binary, which includes:

- Parsing the Mach-O header
- Disassembling the binary code
- Extracting embedded strings, function/methods names, etc.

Once its analysis is complete, Hopper will automatically display the disassembled code at the binary's entry point (extracted from the `LC_MAIN` load command in the Mach-O header).

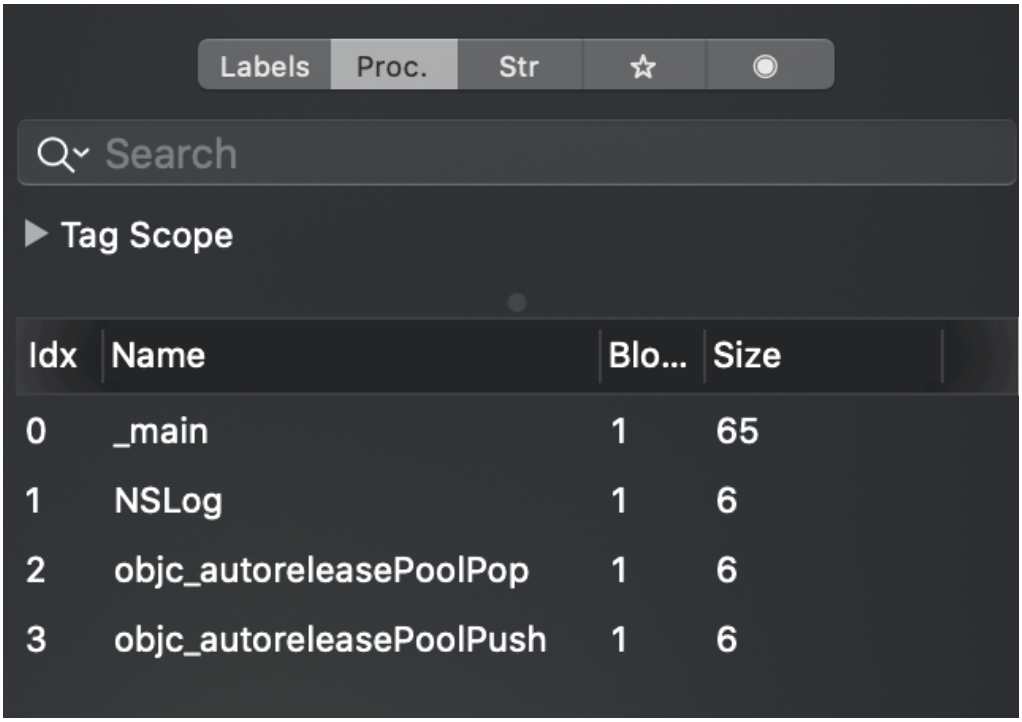
...but first, let's look at various information and options within the Hopper UI.

On the far right is the "inspector" view. This is where Hopper displays general information about the binary being analyzed, including the type of binary (Mach-O), architecture/CPU (Intel x86_64), and calling convention (System V):



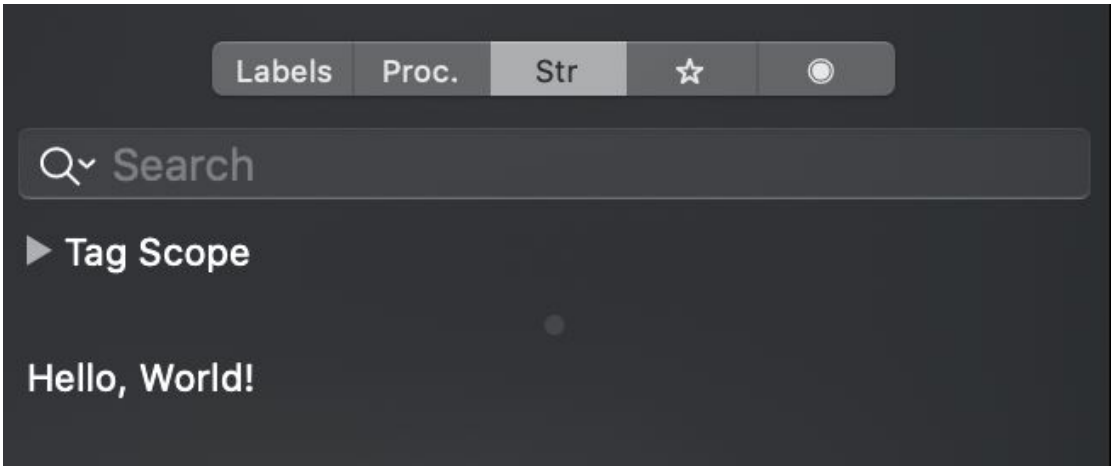
basic file information
(Hopper.app)

On the far left, is a segment-selector that can toggle between various views related to symbols and strings in the binary. For example, the "Proc." view shows procedures that Hopper has identified during its analysis. This includes functions and methods from the original source code, as well as APIs that the code invokes. For example, in our "hello world" binary, Hopper has identified the `main` function and the call to Apple's `NSLog` API:



procedure view
(Hopper.app)

The “Str” view shows the embedded strings that Hopper has extracted from the binary. In our simple binary, the only embedded string is “Hello, World!”:



(embedded) strings view
(Hopper.app)

Before diving into the disassembly, it’s wise to peruse the extracted procedure names and embedded strings as they are often an invaluable source of information about the (possible) capabilities of the malware. Moreover, they can guide analysis efforts. Does a procedure name or embedded string look of interest? Simply click on it, and Hopper will show you exactly where it’s referenced in the binary.

By default, Hopper will automatically display the disassembly of the binary’s entry point (often the `main` function). Here’s the disassembly of the main function in its entirety:

```
        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

        ; Variables:
        ;    var_4: int32_t, -4
        ;    var_8: int32_t, -8
        ;    var_10: int64_t, -16
        ;    var_18: int64_t, -24


                     _main:
0x0000000100000f20          push       rbp
0x0000000100000f21          mov        rbp, rsp
0x0000000100000f24          sub        rsp, 0x20
0x0000000100000f28          mov        dword [rbp+var_4], 0x0
0x0000000100000f2f          mov        dword [rbp+var_8], edi
0x0000000100000f32          mov        qword [rbp+var_10], rsi
0x0000000100000f36          call       imp___stubs__objc_autoreleasePoolPush     ; objc_autoreleasePoolPush
0x0000000100000f3b          lea        rcx, qword [cfstring_Hello__World_]       ; @"Hello, World!"
0x0000000100000f42          mov        rdi, rcx                                  ; argument "format" for method imp___stubs__NSLog
0x0000000100000f45          mov        qword [rbp+var_18], rax
0x0000000100000f49          mov        al, 0x0
0x0000000100000f4b          call       imp___stubs__NSLog                        ; NSLog
0x0000000100000f50          mov        rdi, qword [rbp+var_18]                   ; argument "pool" for method imp___stubs__objc_autoreleasePoolPop
0x0000000100000f54          call       imp___stubs__objc_autoreleasePoolPop      ; objc_autoreleasePoolPop
0x0000000100000f59          xor        eax, eax
0x0000000100000f5b          add        rsp, 0x20
0x0000000100000f5f          pop        rbp
0x0000000100000f60          ret
                         ; endp
```

"Hello World" disassembled
(Hopper.app)

...fairly standard (dis)assembly. However, Hopper does provide helpful annotations such as identifying function names (i.e. mapping `imp__stubs__NSLog` to `NSLog`). Moreover, as it also generally understands API prototypes, it will identify function/method parameters and annotate the assembly code as such.

For example, for the assembly code at address `0x0000000100000f42` which moves the `rcx` register (a pointer to our "Hello, World!" string) into `rdi`, Hopper has identified this as initializing the arguments for a call to `NSLog` (a few lines later).

Various components within the disassembly are actually pointers to data elsewhere in the binary. For example, the assembly code at `0x0000000100000f3b` (`lea rcx, qword [cfstring_Hello__World_]`) is loading the address of the "Hello, World!" string into the `rcx` register.

Hopper is smart enough to identify the `cfstring_Hello__World_` variable as a pointer and thus annotate the assembly code with the value (bytes) of the string ("Hello, World!"). Moreover, if one double-clicks on any pointer, Hopper will jump to the pointer's address. For example, clicking twice on the `cfstring_Hello__World_` variable in the disassembly takes you to the string object at address `0x0000000100001008`:

```
01  cfstring_Hello__World_: ; "Hello, World!"
02  0x0000000100001008  dq  0x0000000100008008,
03  0x0000000100001010  dq  0x00000000000007c8,
04  0x0000000100001018  dq  0x0000000100000fa2,
05  0x0000000100001020  dq  0x000000000000000d
```

This string object (of type `CFConstantString`) itself contains pointers ...and double-clicking on those again takes you to the specified address.

For example, at offset `+0x0` is a pointer with the value of `0x0000000100008008`. Double-clicking on this value takes us to a symbol labeled `___CFConstantStringClassReference` (the class type of the string object). While at offset `+0x10` is a pointer to the actual bytes of the string (found at `0x0000000100000fa2`):

```
01  aHelloWorld:
02  0x0000000100000fa2  db  "Hello, World!", 0  ; DATA XREF=cfstring_Hello__World_
```

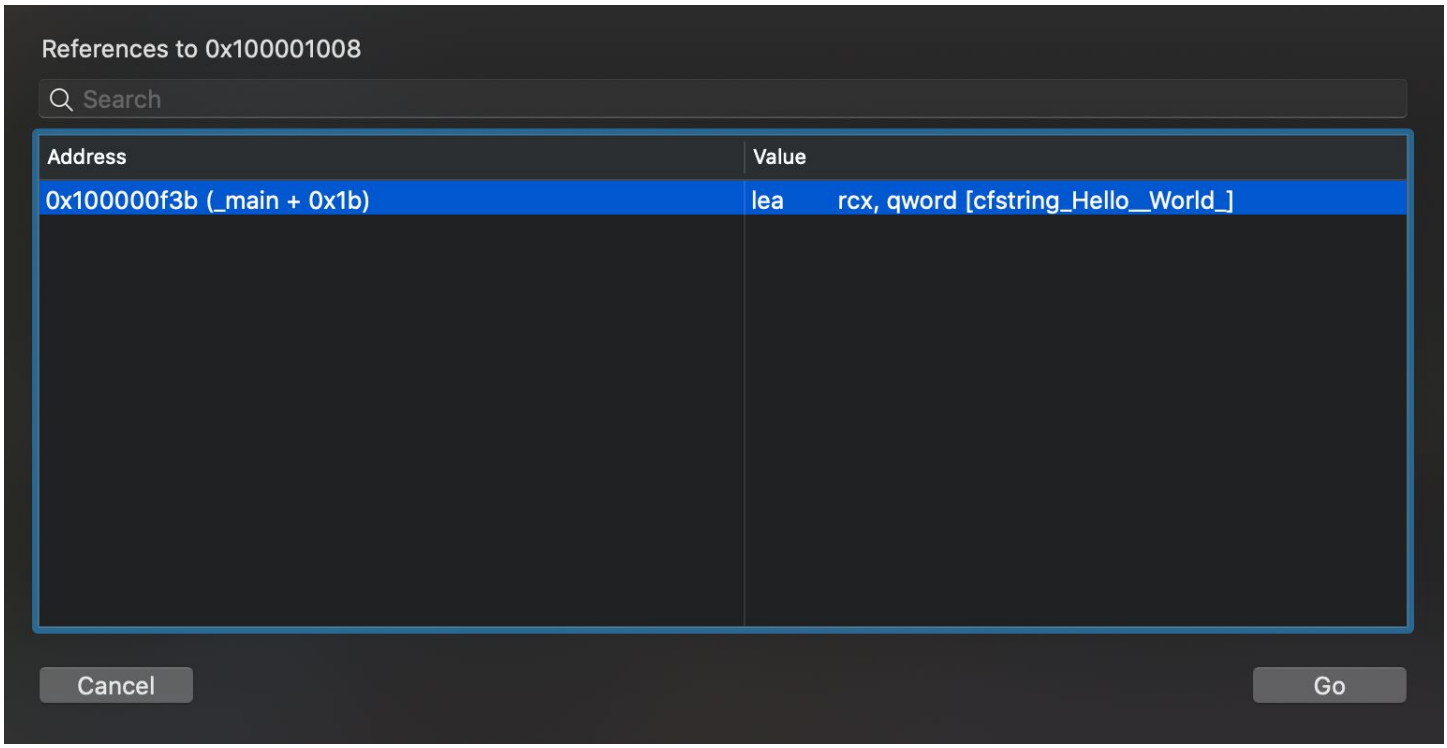
Note that Hopper also tracks (backwards) cross-references! For example, it has identified that the string bytes (at address `0x0000000100000fa2`) are cross-referenced by the `cfstring_Hello__World_` variable. That is to say, the `cfstring_Hello__World_` variable contains a reference to the `0x0000000100000fa2` address.

Such cross-references greatly facilitate static analysis of the binary code. For example, if you notice a string of interest, you can simply ask Hopper where in the code that string is referenced. To view such cross-references, control-click on the address or item and select “References to …” ...or with the address/item selected simply hit “X”.

For example, say we want to see where in disassembly, the “Hello World!” string object is referenced. First we select the string object (at address `0x0000000100001008`), control-click to bring up the context menu, and “References to cfstring_Hello__World”:

cross references
(Hopper.app)

...which brings up the “Cross References” window of that item:



cross reference window
(Hopper.app)

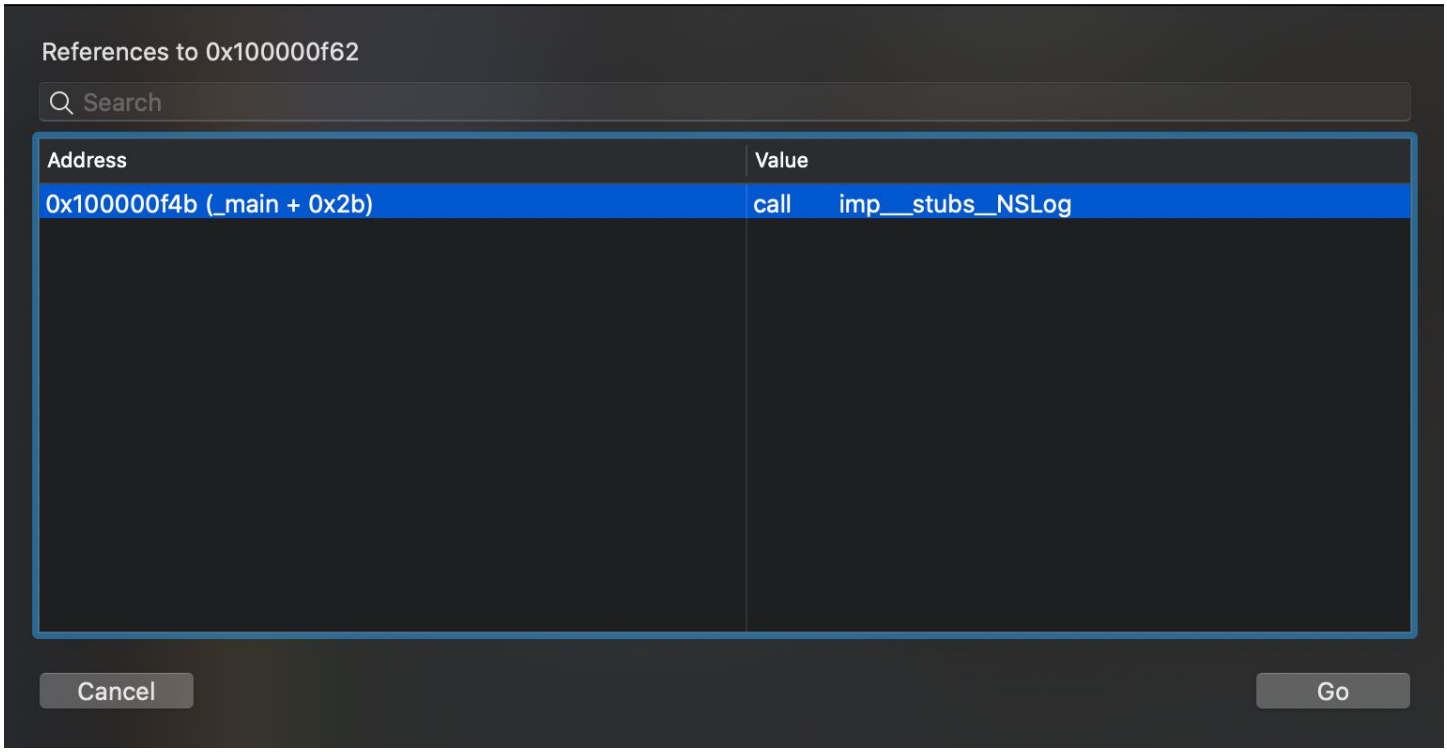
In this example there is only one cross-reference, the code at address `0x0000000100000f3b` (which falls within the `main` function). Click on this to jump to the code in the `main` function, which references the "Hello World" string object:

```
                    _main:
0x0000000100000f20          push        rbp
0x0000000100000f21          mov         rbp, rsp
0x0000000100000f24          sub         rsp, 0x20
0x0000000100000f28          mov         dword [rbp+var_4], 0x0
0x0000000100000f2f          mov         dword [rbp+var_8], edi
0x0000000100000f32          mov         qword [rbp+var_10], rsi
0x0000000100000f36          call        imp___stubs__objc_autoreleasePoolPush
0x0000000100000f3b          lea         rcx, qword [cfstring_Hello__World_]
0x0000000100000f42          mov         rdi, rcx
0x0000000100000f45          mov         qword [rbp+var_18], rax
0x0000000100000f49          mov         al, 0x0
0x0000000100000f4b          call        imp___stubs__NSLog
```

"Hello World" (cf)string
(Hopper.app)

Hopper also creates cross-references for functions, methods, and API calls so that you

can easily determine where in code these are invoked. For example, we can see via the following "Cross References" window that the `NSLog` API is invoked within the `main` function, specifically at `0x0000000100000f4b`:



cross reference window
(Hopper.app)

Cross-references greatly facilitate analysis and can efficiently lead to an understanding of the binary's functionality or capabilities. For example, when analyzing a suspected malware sample, one can locate APIs of interest (perhaps Apple's networking methods that may reveal a connection to a C&C server?) in Hoppers "Proc" view. From this view follow their cross-references to quickly locate relevant code to fully understand how these APIs are being used.

When bouncing around in Hopper (for example following pointer or cross-references), one often wants to quickly return to a previous spot of analysis. Luckily the "esc" key is mapped to "back" and will take you back to where you just were, or further (on multiple key presses).

So far we've stayed in Hopper's default display mode: "Assembly Mode." As the name suggests, this mode displays (dis)assembly of binary code. The display mode can be toggled via a segment control found in Hopper's main toolbar:

display modes
(Hopper.app)

Hopper's supported display modes include:

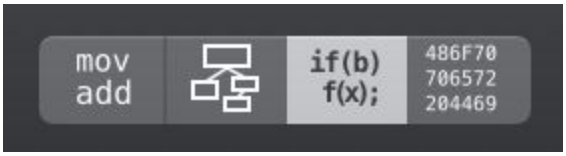
- Assembly mode:
  The standard disassembly mode, in which Hopper *"prints the lines of assembly code, one after the other."* [15]

- Control Flow Graph mode:
  This mode breaks down procedures (e.g. functions) into condition blocks and illustrates the control flow between them.

- Pseudo-Code mode:
  This is Hopper's decompiler mode, in which a "source-code like" or pseudo-code representation is generated.

- Hex mode:
  This mode shows the raw hex bytes of the binary, which is about as low-level as you can get!

Of the four display modes, the pseudo-code (decompiler) mode is arguably the most powerful. To enter this mode, first select a procedure, then click on the 3rd button in the Display Modes segment control:



display modes: decompilation
(Hopper.app)

This will instruct Hopper to decompile the code in the procedure in order to generate a pseudo-code representation of the binary code. For our simple example "Hello World" program, it does a lovely job:

```
int _main(int arg0, int arg1) {
    var_18 = objc_autoreleasePoolPush();
    NSLog(@"Hello, World!");
    objc_autoreleasePoolPop(var_18);
    return 0x0;
}
```

...it almost looks exactly like the original source code:

```
01  #import <Foundation/Foundation.h>
02
03  int main(int argc, const char * argv[]) {
04     @autoreleasepool {
05         // insert code here...
06         NSLog(@"Hello, World!");
07     }
08     return 0;
09  }
```

Apple's "Hello World"

...thus, making the binary analysis (of this trivial binary) a breeze!

This wraps up our overview of the Hopper reverse-engineering tool. While brief, it provides the basics to begin reversing Mach-O binaries!

📝 Note:

For a more comprehensive "how to" on using and understanding Hopper, check out the application's official tutorial:

https://www.hopperapp.com/tutorial.html [16]

Up Next

Armed with a solid understanding of static analysis techniques, ranging from basic file type identification to advanced decompilation, we're now ready to turn our attention to

methods of dynamic analysis. As we'll see, such dynamic analysis often provides a more efficient means of performing malware analysis.

Ultimately though, static and dynamic analysis are complementary; their combination provides the ultimate analysis approach.

References

1. “Hacker Disassembling Uncovered”
https://www.amazon.com/Hacker-Disassembling-Uncovered-Kris-Kaspersky/dp/1931769648

2. “Reversing: Secrets of Reverse Engineering”
https://www.amazon.com/Reversing-Secrets-Engineering-Eldad-Eilam/dp/0764574817

3. “x86 assembly language”
https://en.wikipedia.org/wiki/X86_assembly_language

4. objc_msgSend function
https://developer.apple.com/documentation/objectivec/1456712-objc_msgsend

5. “Modern Objective-C Exploitation Techniques”
http://www.phrack.org/issues/69/9.html

6. “The Objective-C Runtime: Understanding and Abusing”
http://www.phrack.org/issues/66/4.html

7. “Sofacy’s ‘Komplex’ OS X Trojan”
https://unit42.paloaltonetworks.com/unit42-sofacys-komplex-os-x-trojan/

8. NSData’s dataWithContentsOfURL: method
https://developer.apple.com/documentation/foundation/nsdata/1547245-datawithcontentsofurl

9. “TEST (x86 instruction)”
https://en.wikipedia.org/wiki/TEST_(x86_instruction)

10. “Lazarus Group Goes 'Fileless'”
https://objective-see.com/blog/blog_0x51.html

11. “kIOMasterPortDefault”
https://developer.apple.com/documentation/iokit/kiomasterportdefault?language=objc

12. “IOServiceMatching”
https://developer.apple.com/documentation/iokit/1514687-ioservicematching?language=objc

13. “IOServiceGetMatchingService”
https://developer.apple.com/documentation/iokit/1514535-ioservicegetmatchingservice

?language=objc

14. "IORegistryEntryCreateCFProperty"
https://developer.apple.com/documentation/iokit/1514293-ioregistryentrycreatecfproperty?language=objc

15. "CFStringGetCString"
https://developer.apple.com/documentation/corefoundation/1542721-cfstringgetcstring?language=objc

16. Hopper
https://www.hopperapp.com/

17. Hopper Tutorial
https://www.hopperapp.com/tutorial.html