CyberSaint

SECURITY

Safely Reverse Engineer macOS Malware: Techniques and Precautions

Author: Sunny thakur

Table of Contents

Introduction

While resources for learning malware analysis and reverse engineering are plentiful for Windows platforms and PE files, there’s a noticeable lack of comprehensive guides tailored to macOS malware and its unique analysis techniques. This project aims to fill that gap, providing a structured approach for those eager to dive into macOS malware analysis.

Throughout this journey, you'll set up a secure analysis environment, acquire essential tools, and find sample files to practice on. We’ll walk through a sample file together, using step-by-step instructions to learn native macOS tools and techniques. By the end, you'll be able to deconstruct a file’s behavior, identifying critical Indicators of Compromise (IoCs) for detection purposes. Along the way, you’ll master both static and dynamic analysis methods—and hopefully, enjoy the process too!

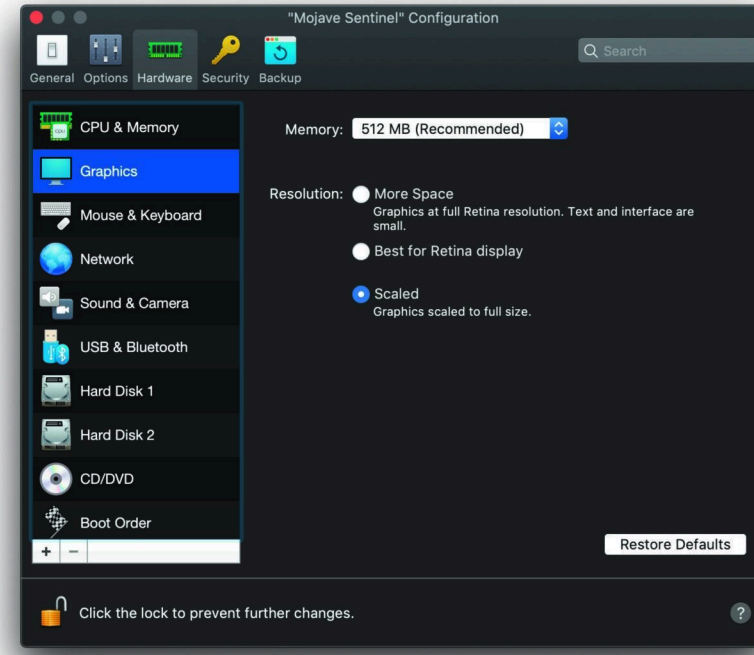
Let’s start!

Part One

How To Set Up A Safe Environment To Test Malware

To safely test malware, you'll need virtualization software to run a guest operating system isolated from your primary system. On macOS, you have three main options: VirtualBox, Parallels, and VMWare. Any of these will work, so choose the one that suits you best, and follow the documentation to set up a macOS Virtual Machine. You can use any recent macOS version, though this tutorial will reference a Parallels Desktop VM running macOS 10.14.3.

Once your virtual machine is set up, take some time to review the configuration options for your guest OS. Proper configuration is essential to ensure that your environment is both secure and functional for malware analysis.



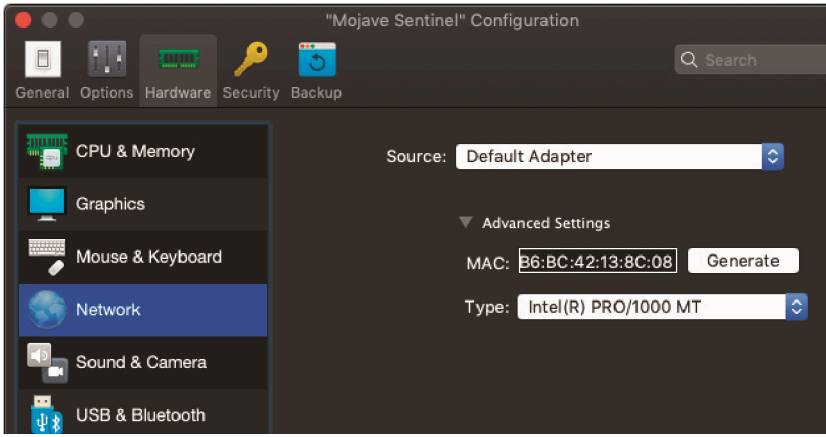
Configuring Your Virtual Machine for Malware Analysis

To ensure your virtual machine (VM) runs smoothly, a few adjustments are necessary. First, VMs can sometimes lag, so allocate enough resources. Set the RAM to at least 2GB, although 4GB is recommended for smoother performance. Additionally, increase the graphics memory to a minimum of 512MB.

Some malware attempts to detect if it's running in a virtual environment and may alter its behavior accordingly. While this tutorial won't cover anti-VM evasion techniques, one helpful trick is to change the default MAC address of your VM:

- For **Parallels**, change the MAC address to avoid the "00-1C-42" prefix.
- For **VirtualBox**, use any MAC address other than those beginning with "08-00-27."
- For **VMWare**, avoid prefixes "00-50-56," "00-0C-29," and "00-05-69."

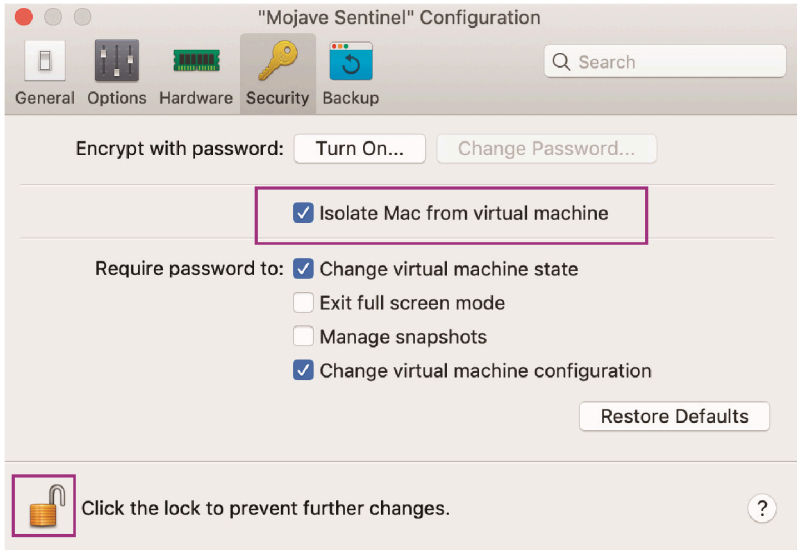
These tweaks will help you establish a robust environment for testing malware without detection issues.



Isolate Your macOS Guests!

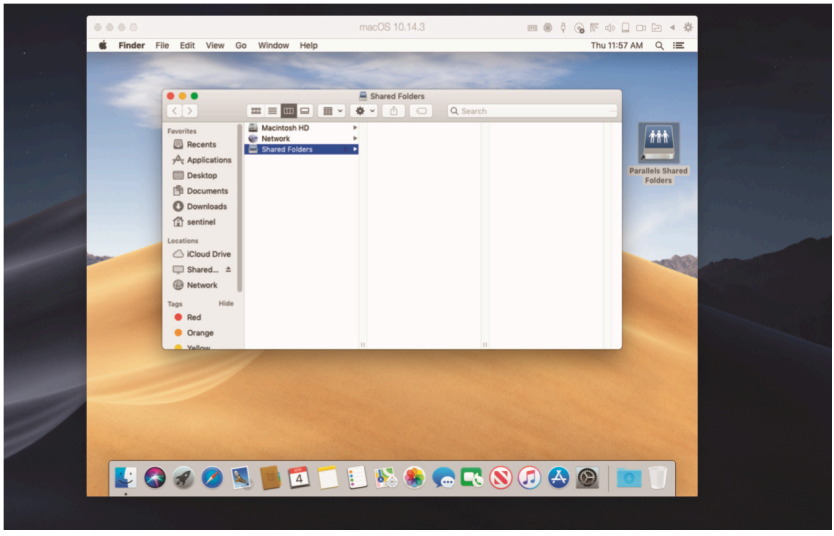
Isolation is crucial to ensure that malware cannot affect your host machine. To achieve this, disable all shared drives and folders between the VM (guest) and your main computer (host), including any backup locations. This prevents the malware from having any path to escape the VM. For additional safety, avoid sharing the clipboard as well.

The exact steps vary depending on the virtualization software you're using and its version, but look for isolation settings in the security options. For example, in Parallels Desktop 14, you can find these controls in the **Security** tab. Proper isolation is key to maintaining a safe testing environment.



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.



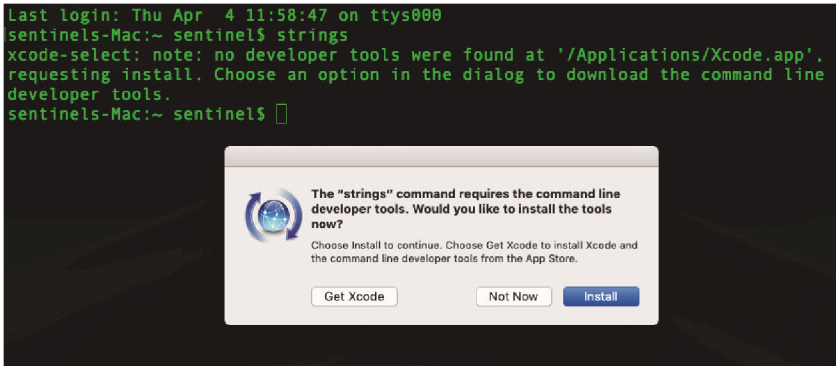
Isolation is essential, but what if you want to offload some screenshots or output files for future reference? In that case, use the guest's browser and a free service like wetransfer.com to email compressed and password-protected files to yourself.

Tools For Testing Malware On macOS

From this point onward, assume that all actions will take place within your isolated VM environment. Take a moment to adjust any **System Preferences** or **Application Preferences** to suit your workflow—especially in the Terminal, as we'll be using it extensively.

On the command line, use the strings utility, which should prompt you to install Apple's command line tools. Follow the prompts to complete the installation.

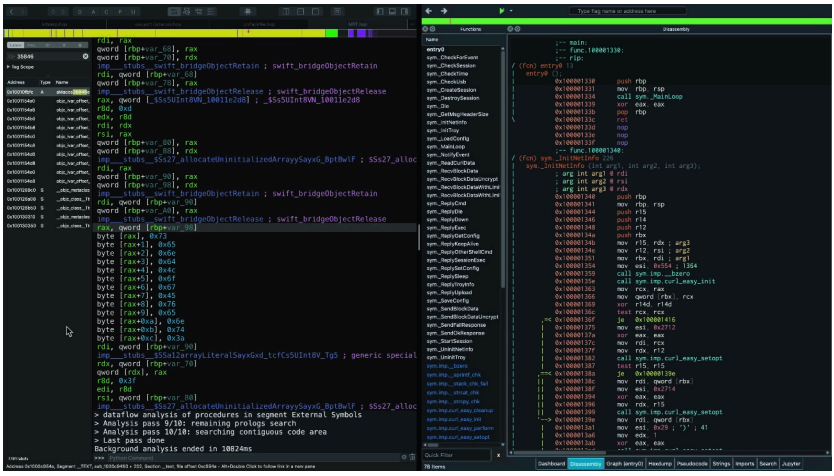
.



With that, you now have all the essential tools needed for learning macOS malware analysis and reverse engineering! These include:

- A string decoder (strings)
- File analysis utilities (file, nm, xattr, mdls, and others)
- Hex editors (hexdump, od, xxd)
- A static disassembler (otool)
- A debugger, memory reader, and dynamic disassembler (lldb)

And the best part? These tools are free and readily available!



While professional malware analysts often use advanced tools like Hopper, Cutter, Radare2, Floss, and other community-created utilities, we'll focus on built-in tools for now. Why? As beginners, it's essential to grasp the core concepts without the complexities that professional tools can introduce. These fundamentals will lay a strong foundation before diving into more advanced tools.

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!

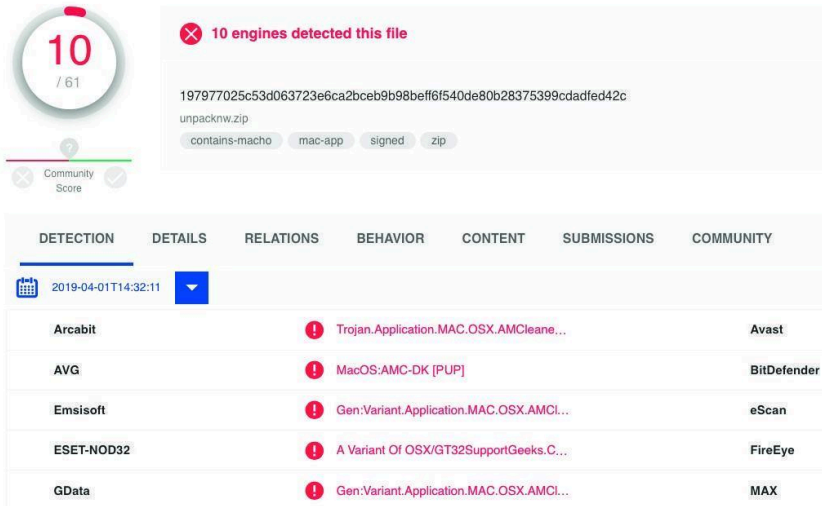
Let's set up a working directory on our VM guest, where we'll save our samples and do all our work. Something like:

```
$ 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.



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
197977025c53d063723e6ca2bceb9b98beff6f540de80b28375399
c 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 lets inflate it and see what we have:

```
$ unzip malware01
```

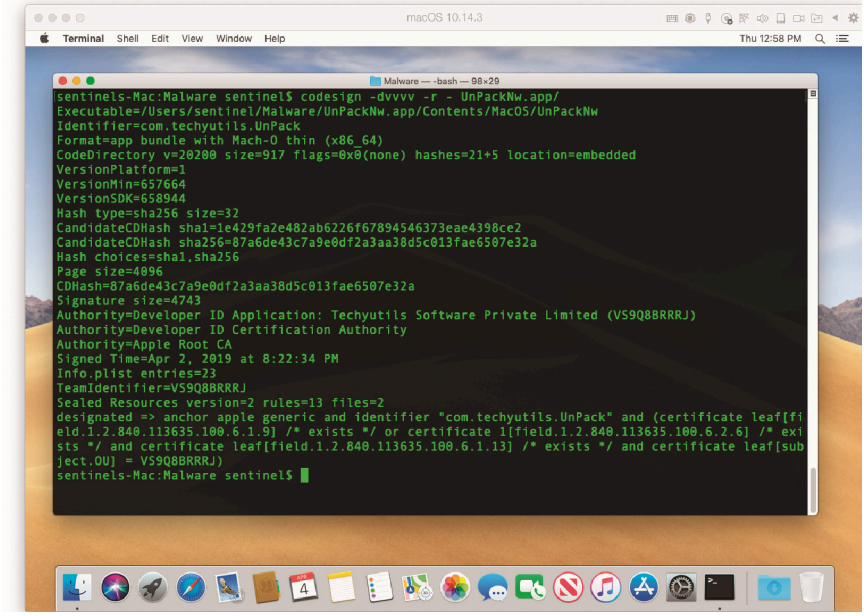
The output from Terminal shows us that it's a macOS application bundle with a regular hierarchy.

```
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/
```



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:

```
$ spctl --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
```

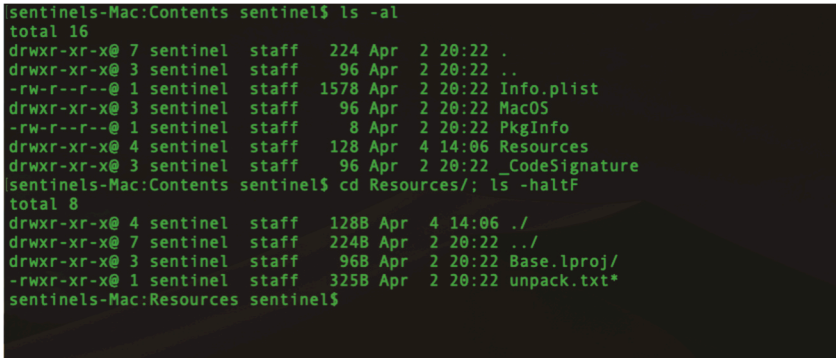
Note that the output includes "**CFBundleIdentifier**", the bundle identifier from the codesign utility, so you can get that for your list of IoCs here if you are dealing with a sample that isn't codesigned.

```
=> "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.

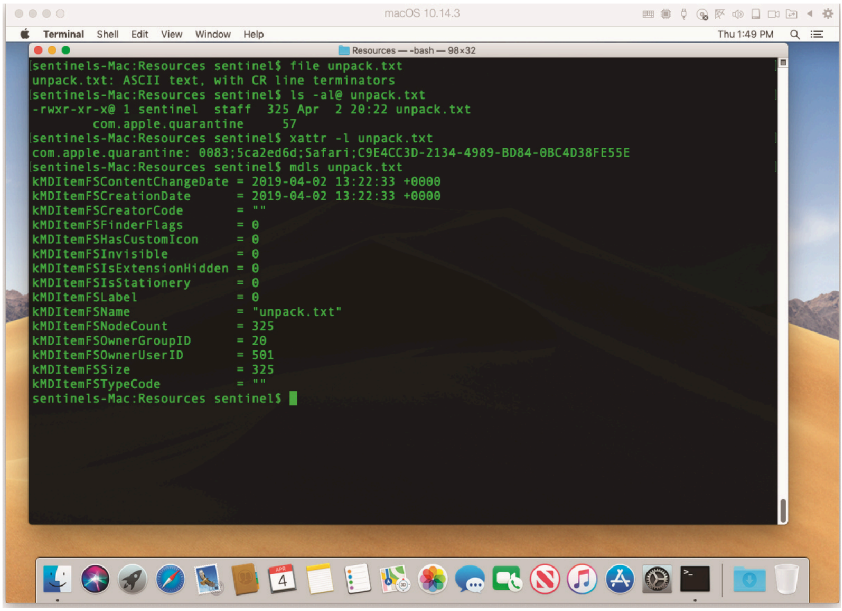


What's that "unpack.txt" file, and why does it have an asterisk after it? Let's collect more details about it before we peek inside.

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.

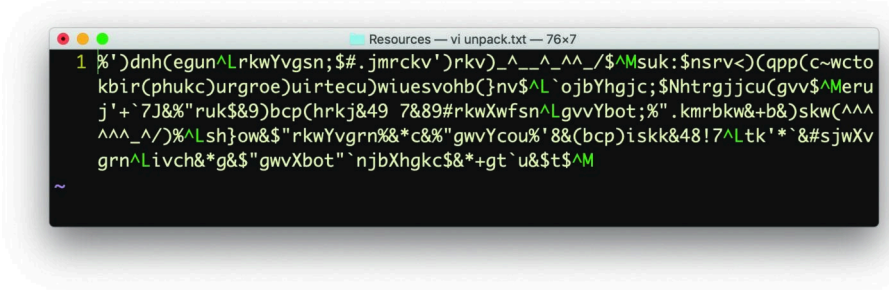


As the image above shows, we can use both **xattr -l** and **ls -al@** to list a file's extended attributes and permissions.

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**.



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 established a secure testing environment to safely analyze macOS malware, explored sources for obtaining malware samples, and examined techniques to inspect application bundles and read file metadata. In our current sample, we've uncovered something intriguing and obfuscated. But the question remains: is it malicious? And if so, how can we decode it?

In Part Two, we'll continue our journey into macOS reverse engineering by diving deeper into static analysis techniques for Mach-O executables, exploring Mach-O disassembly, and further enhancing our reverse engineering skills.

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$
```

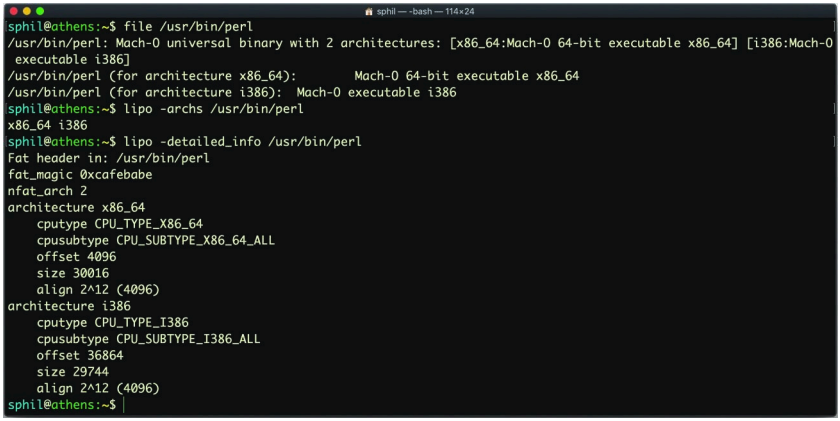
There's a single binary as expected. Let's run file on it and see what it says:

```
$ file UnPackNw
```

```
MacOS — -bash — 74×23
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.



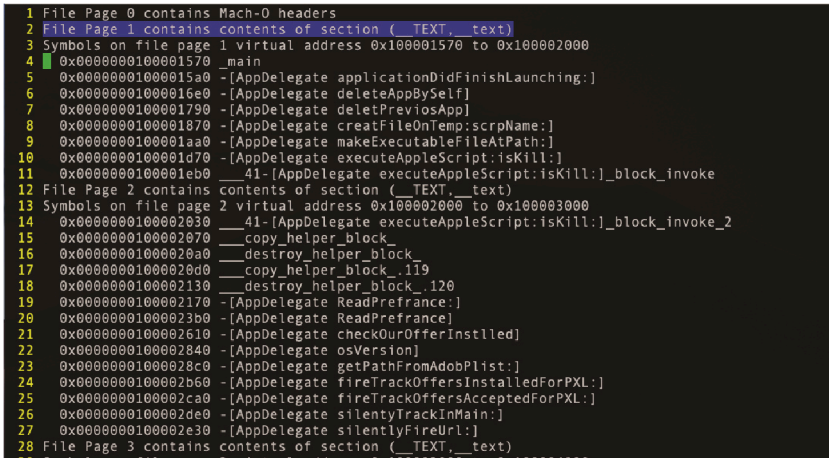
If you find yourself dealing with a “fat” binary, you can easily use the **lipo** tool to extract the Mach-O architecture, but we won’t be needing to do that in this tutorial.

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
```

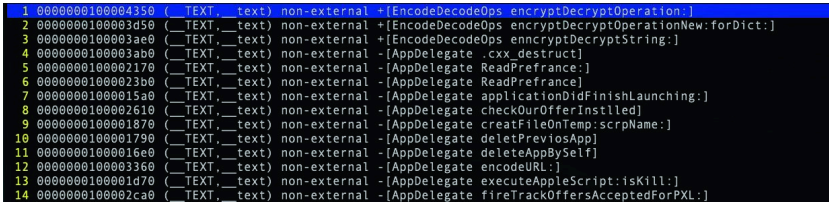
A Mach-O binary contains a number of segments, which are in turn composed of sections. For the purposes of this tutorial, we only need to know that the **__TEXT** segment contains the **__text** section, which contains all the executable functions and methods. A couple of good intros on this topic, which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



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
```



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.

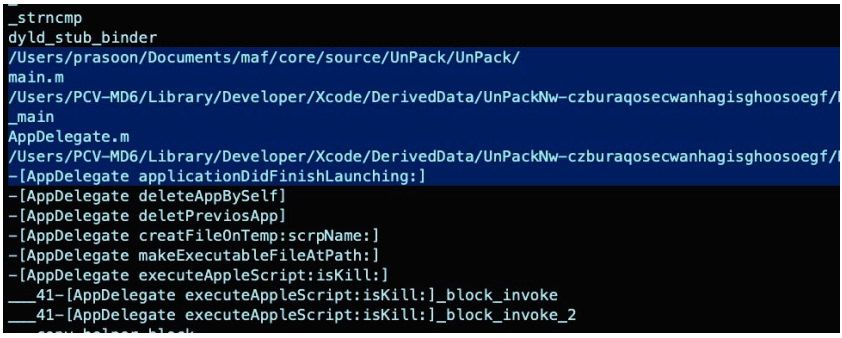
The Power of Pulling Strings

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.



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
```

```
imp 0x1000016e0 -[AppDelegate deleteAppBySelf]
imp 0x100001790 -[AppDelegate deletPreviosApp]
imp 0x100001870 -[AppDelegate creatFileOnTemp:scrpName:]
imp 0x100001aa0 -[AppDelegate makeExecutableFileAtPath:]
imp 0x100001d70 -[AppDelegate executeAppleScript:isKill:]
imp 0x1000028c0 -[AppDelegate getPathFromAdobPlist:]
imp 0x100002b60 -[AppDelegate fireTrackOffersInstalledForPXL:]
imp 0x100002ca0 -[AppDelegate fireTrackOffersAcceptedForPXL:]
imp 0x100002de0 -[AppDelegate silentyTrackInMain:]
imp 0x100002e30 -[AppDelegate silentlyFireUrl:]
imp 0x100003060 -[AppDelegate getEncodedURL:searchString:]
imp 0x100003360 -[AppDelegate encodeURL:]
imp 0x100003970 -[AppDelegate webViewTrack]
imp 0x1000039a0 -[AppDelegate setWebViewTrack:]
imp 0x100003ae0 +[EncodeDecodeOps enncryptDecryptString:]
imp 0x100003d50 +[EncodeDecodeOps encryptDecryptOperationNew:forDict:]
imp 0x100004350 +[EncodeDecodeOps encryptDecryptOperation:]
```

Most usefully, we can obtain the disassembly with:

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

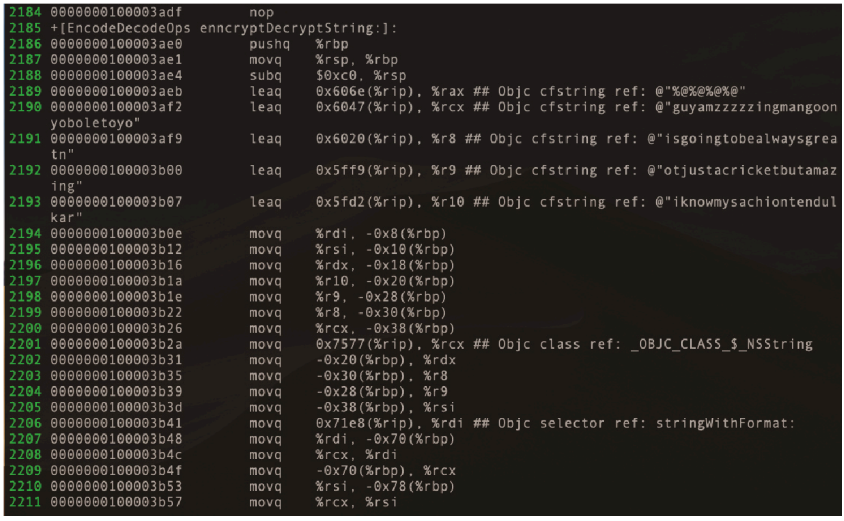
In the disassembly, search for the name of our obfuscated text file, ‘unpack’:

```
0000000100001ff        movq    0x96ea(%rip), %rsi ## Objc selector ref: mainBundle
0000000100001606       movq    %rax, %rdi
0000000100001609       callq   *0x7a99(%rip) ## Objc message: +[NSBundle mainBundle]
000000010000160f       leaq    0x7daa(%rip), %rsi ## Objc cfstring ref: @"unpack"
0000000100001616       leaq    0x7dc3(%rip), %rdi ## Objc cfstring ref: @"txt"
000000010000161d       movq    0x96d4(%rip), %rcx ## Objc selector ref: pathForResource:ofType
:
0000000100001624       movq    %rdi, -0x30(%rbp)
0000000100001628       movq    %rax, %rdi
000000010000162b       movq    %rsi, -0x38(%rbp)
000000010000162f       movq    %rcx, %rsi
0000000100001632       movq    -0x38(%rbp), %rdx
0000000100001636       movq    -0x30(%rbp), %rcx
000000010000163a       callq   *0x7a68(%rip) ## Objc message: -[%rdi pathForResource:ofType:]
0000000100001640       movl    $0x4, %r8d
0000000100001646       movl    %r8d, %ecx
0000000100001649       xorl    %r8d, %r8d
000000010000164c       movq    %rax, -0x20(%rbp)
0000000100001650       movq    0x9a51(%rip), %rax ## Objc class ref: _OBJC_CLASS_$_NSString
0000000100001657       movq    -0x20(%rbp), %rdx
000000010000165b       movq    0x969e(%rip), %rsi ## Objc selector ref: stringWithContentsOfFi
le:encoding:error:
0000000100001662       movq    %rax, %rdi
0000000100001665       callq   *0x7a3d(%rip) ## Objc message: +[NSString stringWithContentsOfF
ile:encoding:error:]
000000010000166b       movq    %rax, -0x28(%rbp)
000000010000166f       movq    0x9a3a(%rip), %rax ## Objc class ref: EncodeDecodeOps
0000000100001676       movq    -0x28(%rbp), %rdx
000000010000167a       movq    0x9687(%rip), %rsi ## Objc selector ref: enncryptDecryptString:
0000000100001681       movq    %rax, %rdi
0000000100001684       callq   *0x7a1e(%rip) ## Objc message: +[EncodeDecodeOps enncryptDecryp
tString:]
```

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:

That takes us to Line 2185:

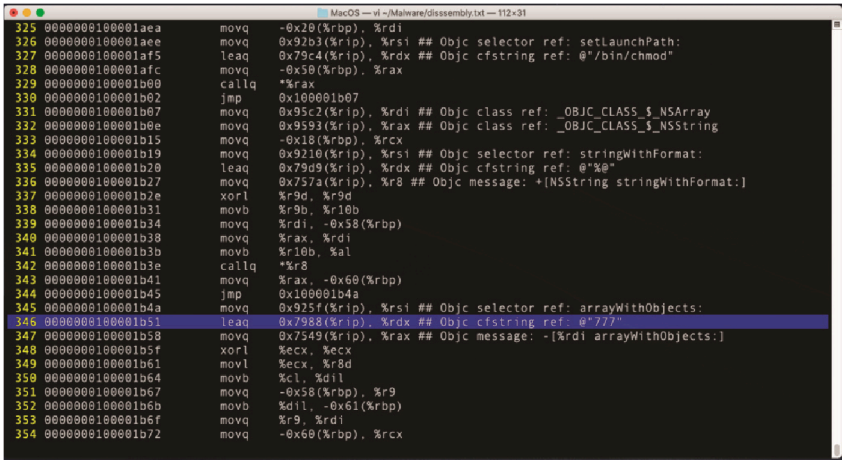


Lines 2190 to 2193 are revealing. We’re starting to get closer to solving the mystery of our encrypted text file. At this point, I’d probably jump into Cutter or Hopper and see how this looks in pseudocode, but the assembly already suggests to us that this is going to iterate over some hardcoded strings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

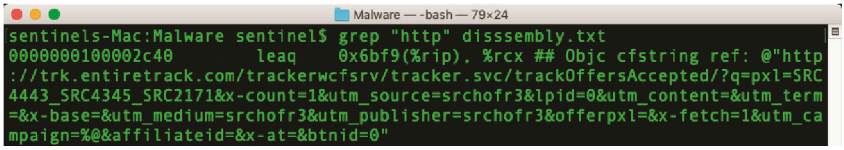
Compiling Indicators of Compromise

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:



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.



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 behavior 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 encrypt Decrypt String:]**. 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.

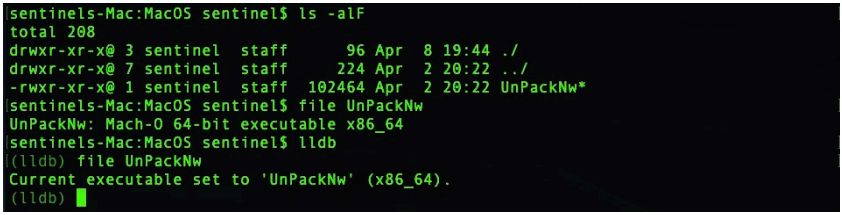
```
$ cd ~/Malware/UnPackNw.app/Contents/MacOS
```

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 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**:



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.

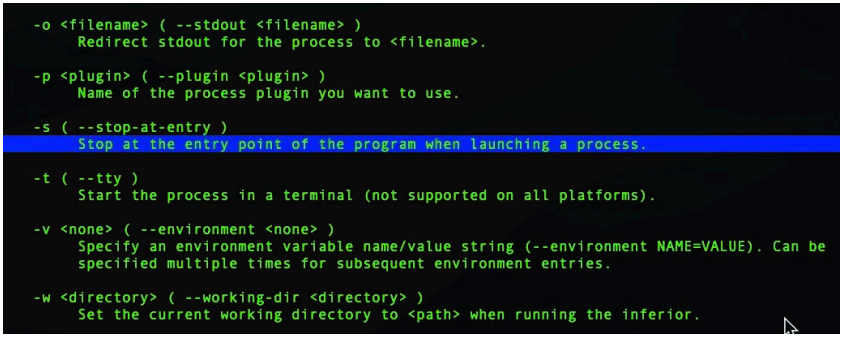
The next step is to launch the malware, but we don't want to just fire the whole thing off and let it do what it wants. We need to control the execution, which we do by using the **process** command. Let's take a moment to see what that does:

```
(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.



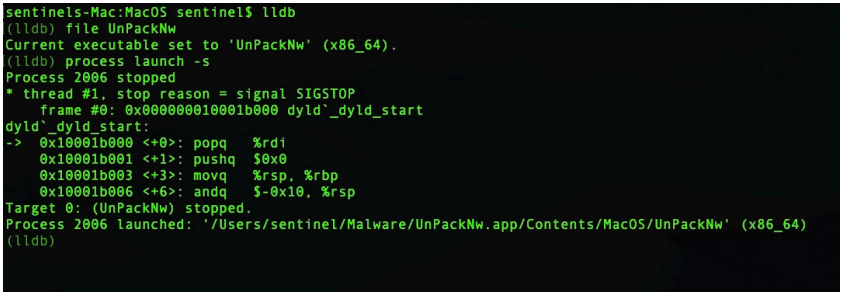
The first entry point should be **dyld_start**, which is when the dynamic linker starts loading any libraries the malware relies on before getting to the binary's own code (recall from Part 2 that we can list dependent libraries with **otool -L**).

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



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 enncryptDecryptString:]"
```

```
(lldb) breakpoint set -n "+[EncodeDecodeOps enncryptDecryptString:]"
Breakpoint 1: where = UnPackNw`+[EncodeDecodeOps enncryptDecryptString:], address = 0x000000010000
3ae0
(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).

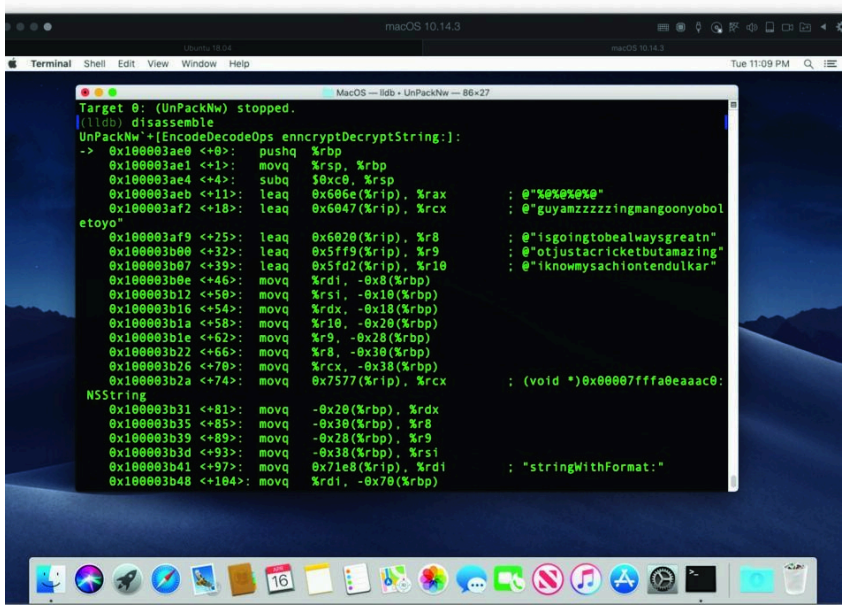
With our breakpoint successfully set, we need to type either **continue** or just the letter **c** to tell the debugger to resume execution until it hits our breakpoint.

```
(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 enncryptDecryptString:]
UnPackNw`+[EncodeDecodeOps enncryptDecryptString:]:
->  0x100003ae0 <+0>:  pushq  %rbp
    0x100003ae1 <+1>:  movq   %rsp, %rbp
    0x100003ae4 <+4>:  subq   $0xc0, %rsp
    0x100003aeb <+11>: leaq   0x606e(%rip), %rax        ; @"%@%@%@%@"
Target 0: (UnPackNw) stopped.
(lldb)
```

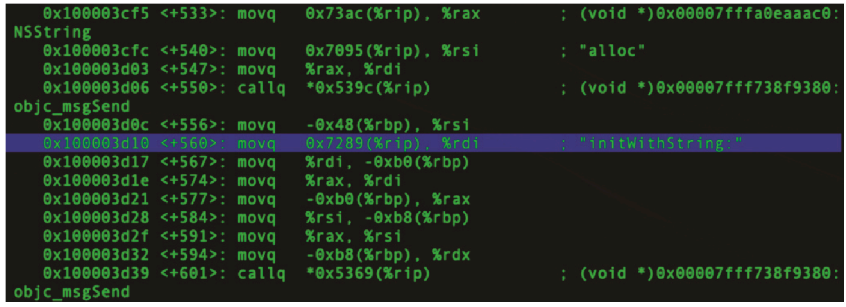
We've stopped at the entry to the function. Let's see a bit more of the disassembly so we can orient ourselves.

(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.



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

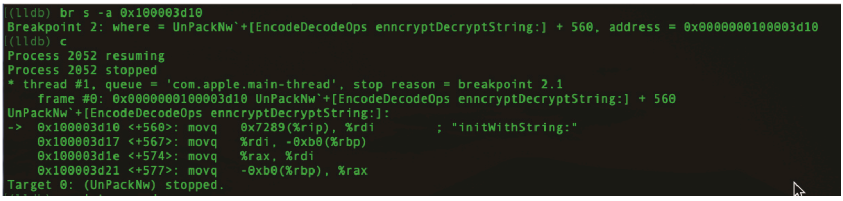


That looks like the method where the code will create a new plain-text string from the encrypted code in **unpack.txt**. We can tell that because it occurs just before

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
```

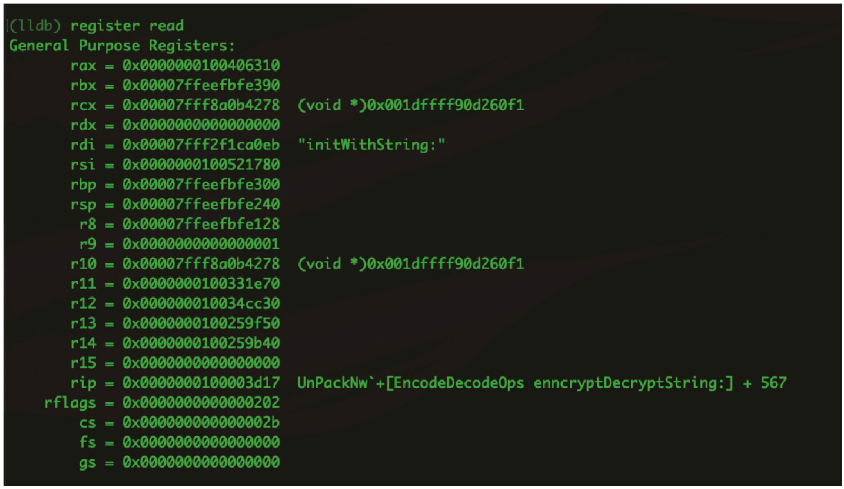


How to Read Registers in LLDB

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
```

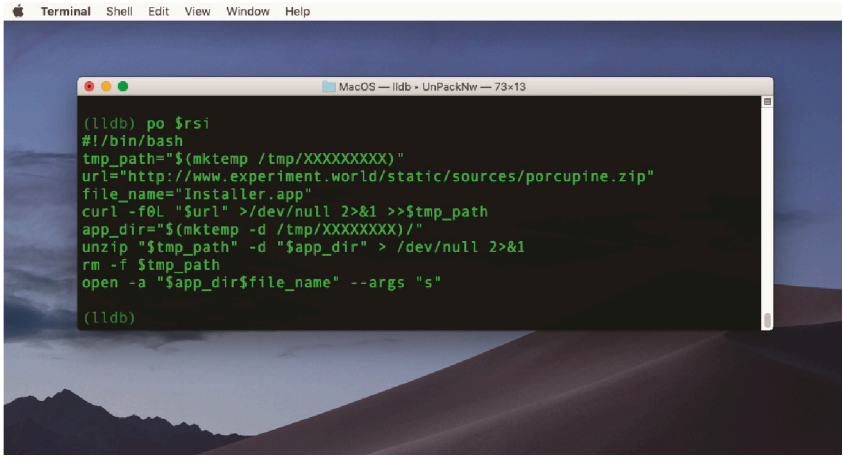


As we're dealing with 64-bit architecture, all our general registers begin with "r": **rax**, **rbx**, **rcx**, and so on.

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
```



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

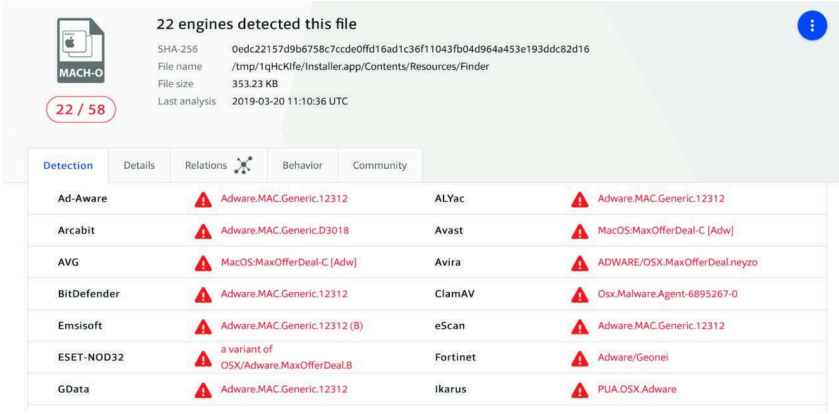
If you want to let the malware just play out the rest of its behaviour, use **continue** again in the debugger. Since we haven’t set any more breakpoints, it’ll either complete its execution or stop on a further call to the decrypt method.

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.



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 **lldb**. 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.