objective-see.com/blog/blog_0x3B.html

Middle East Cyber-Espionage

analyzing WindShift's implant: OSX.WindTail (part 1)

December 20, 2018

📝 👾 Want to play along?

I've shared various `OSX.WindTail` samples (password: infect3d) …don't infect yourself!

In this blog post, we'll analyze the `WindShift` APT group's 1st-stage macOS implant: `OSX.WindTail` (likely variant `A`)

Specifically we'll detail the malware's:

- initial infection vector
- method of persistence
- capabilities
- detection and removal

Background

A few months ago, Taha Karim (head of malware research labs, at Dark Matter) presented some intriguing research at Hack in the Box Singapore.

> "[the APT] targeted specific individuals working in government departments and critical infrastructure across the Middle East"

To me, besides WindShift's targets, the most intriguing aspect of this APT group was (is?) their use of macOS vulnerabilities and custom macOS implants (backdoors). In his talk, Karim provided a good overview of the technique utilized by WindShift to infect macOS computers, and the malware they then installed (`OSX.WindTail.A` , `OSX.WindTail.B` , and `OSX.WindTape`). However, my rather insatiable technical cravings weren't fully satisfied, so I decided to dig deeper!

From the details Karim's talk, I was able to replicate WindShift's macOS exploitation capabilities:

patrick wardle
@patrickwardle

If you want to remotely infect Macs, read this 🍎🤒objective-see.com/blog/blog_0x38…

credit:

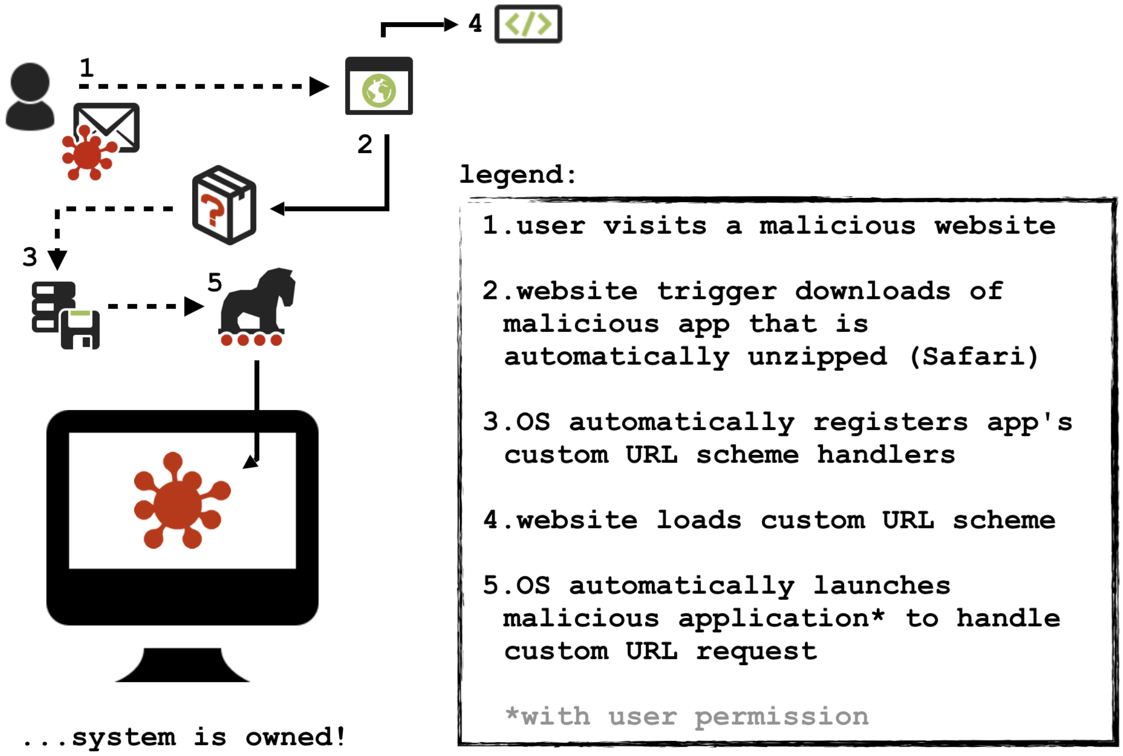
0️⃣ WINDSHIFT APT who is infecting 🍎-users of Middle Eastern .govts
1️⃣ Taha Karim/@GuardedbyGenius for the excellent "The Trails of WINDSHIFT APT" talk at @hitbgsec 🙏

343 6:05 PM - Aug 30, 2018

Remote Mac Exploitation Via Custom URL Schemes
a cyber-espionage campaign infects macs with a novel infection mechanism
objective-see.com

210 people are talking about this

My blog post, “Remote Mac Exploitation Via Custom URL Schemes”, describes the technical details of how WindShift (ab)used custom URL schemes to infect macOS systems. The image below provides a illustrative overview.



legend:

1.user visits a malicious website

2.website trigger downloads of malicious app that is automatically unzipped (Safari)

3.OS automatically registers app's custom URL scheme handlers

4.website loads custom URL scheme

5.OS automatically launches malicious application* to handle custom URL request

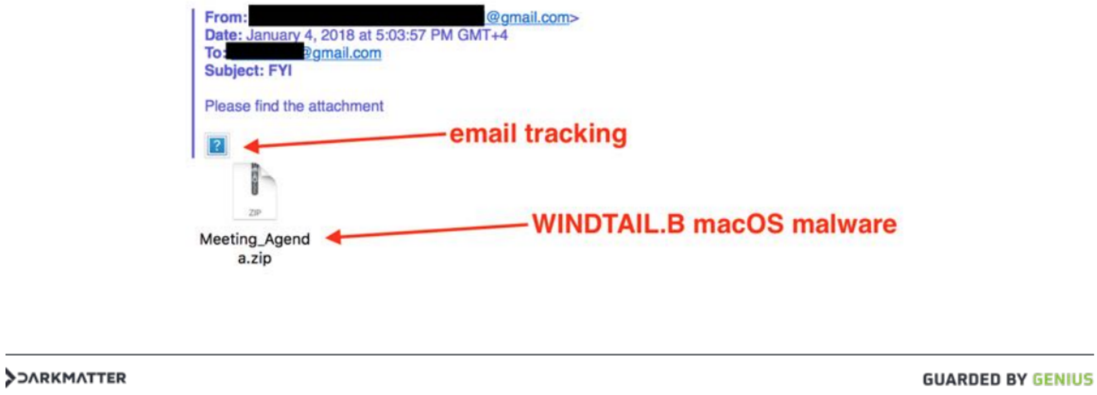
*with user permission

...system is owned!

…however, as the malware samples discussed in Karim's talk were never publicly shared, a full-technical analysis was never available…until now!

Analyzing `OSX.WindTail`

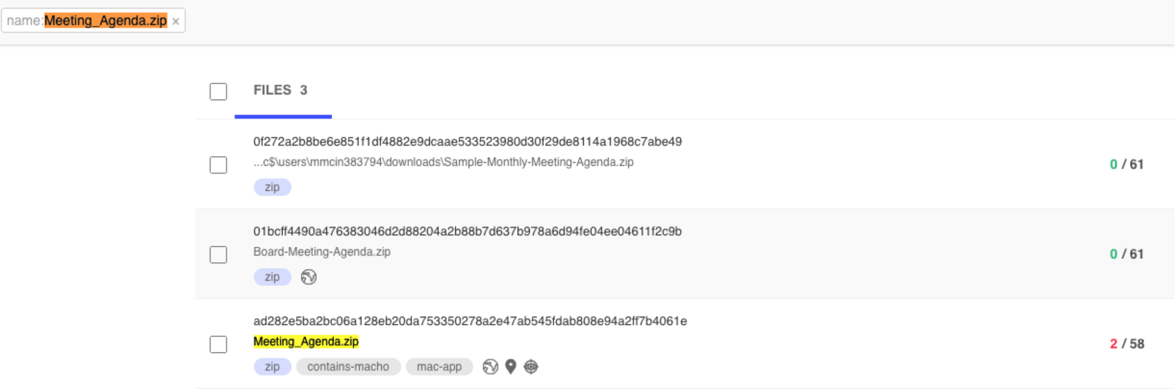
Earlier today, Phil Stokes, uncovered an interesting application on VirusTotal. He noted that in Karim's talk, one of the slides contained a file name: `Meeting_Agenda.zip` … which was identified as by Karim as malware:



From: [redacted]@gmail.com>
Date: January 4, 2018 at 5:03:57 PM GMT+4
To: [redacted]@gmail.com
Subject: FYI

Please find the attachment

On VirusTotal, if we search for files with this name, we find what appears to be a match!

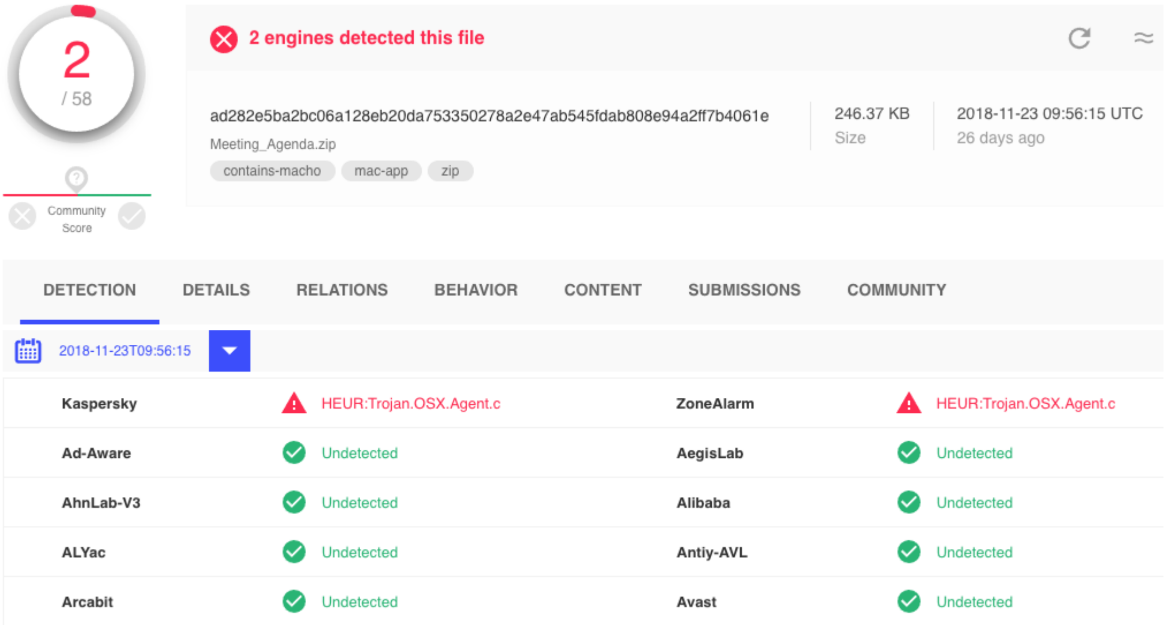


name:Meeting_Agenda.zip ×

□ FILES 3

□	0f272a2b8be6e851f1df4882e9dcaae533523980d30f29de8114a1968c7abe49 ...c$\users\mmcin383794\downloads\Sample-Monthly-Meeting-Agenda.zip zip	0 / 61
□	01bcff4490a476383046d2d88204a2b88b7d637b978a6d94fe04ee04611f2c9b Board-Meeting-Agenda.zip zip	0 / 61
□	ad282e5ba2bc06a128eb20da753350278a2e47ab545fdab808e94a2ff7b4061e Meeting_Agenda.zip zip contains-macho mac-app	2 / 58

The sample (SHA-1: `4613f5b1e172cb08d6a2e7f2186e2fdd875b24e5`) is currently only detected by two anti-virus engines:



2 / 58

Community Score

2 engines detected this file

ad282e5ba2bc06a128eb20da753350278a2e47ab545fdab808e94a2ff7b4061e
Meeting_Agenda.zip
contains-macho mac-app zip

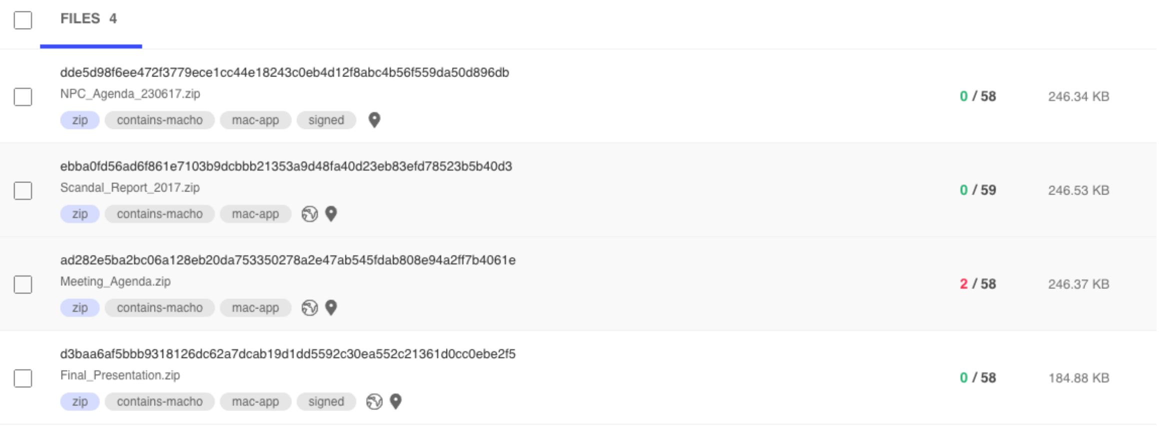
246.37 KB
Size

2018-11-23 09:56:15 UTC
26 days ago

2018-11-23T09:56:15

Kaspersky	HEUR:Trojan.OSX.Agent.c	ZoneAlarm	HEUR:Trojan.OSX.Agent.c
Ad-Aware	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
ALYac	Undetected	Antiy-AVL	Undetected
Arcabit	Undetected	Avast	Undetected

Using the `similar-to:` search modifier, I uncovered three other samples, that are not flagged as malicious by any anti-virus engine!



□ FILES 4

□ dde5d98f6ee472f3779ece1cc44e18243c0eb4d12f8abc4b56f559da50d896db
NPC_Agenda_230617.zip
zip contains-macho mac-app signed
0 / 58 246.34 KB

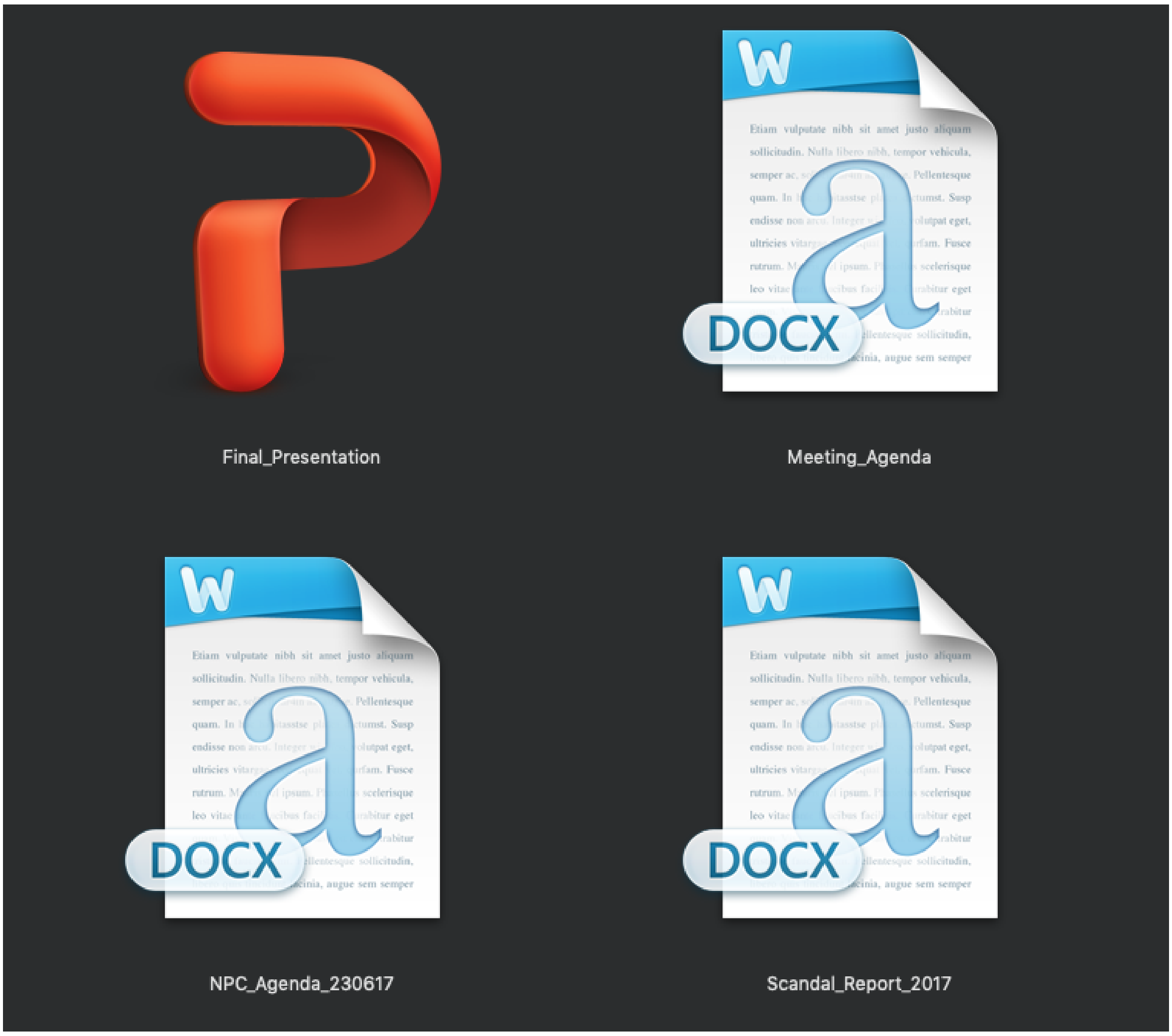
□ ebba0fd56ad6f861e7103b9dcbbb21353a9d48fa40d23eb83efd78523b5b40d3
Scandal_Report_2017.zip
zip contains-macho mac-app
0 / 59 246.53 KB

□ ad282e5ba2bc06a128eb20da753350278a2e47ab545fdab808e94a2ff7b4061e
Meeting_Agenda.zip
zip contains-macho mac-app
2 / 58 246.37 KB

□ d3baa6af5bbb9318126dc62a7dcab19d1dd5592c30ea552c21361d0cc0ebe2f5
Final_Presentation.zip
zip contains-macho mac-app signed
0 / 58 184.88 KB

- `NPC_Agenda_230617.zip1`
  SHA-1: `df2a83dc0ae09c970e7318b93d95041395976da7`
- `Scandal_Report_2017.zip`
  SHA-1: `6d1614617732f106d5ab01125cb8e57119f29d91`
- `Final_Presentation.zip`
  SHA-1: `da342c4ca1b2ab31483c6f2d43cdcc195dfe481b`

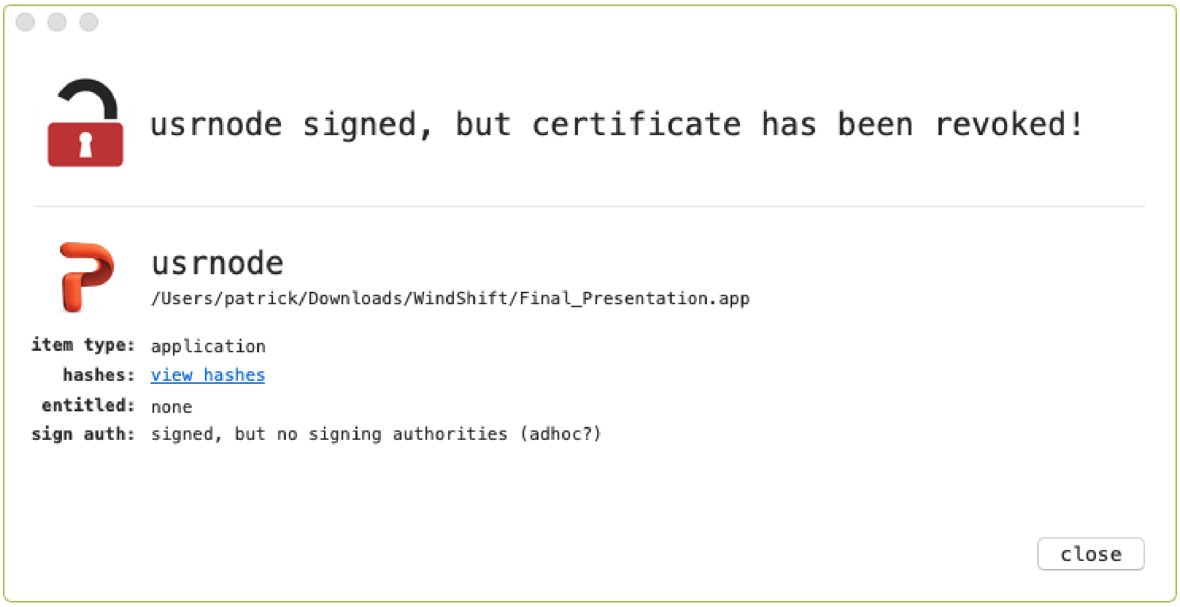
If we download and extract these applications, the uses Microsoft Office icons, likely to avoid raising suspicion:



In his talk, Karim notes, “[the WindShift] attackers gave a backdoor a realistic look by mimicking an Excel sheet icon”.

…the fact that our samples all similarly utilize Microsoft Office icons, is the first (of many) characteristics that lead us to confidently tie these samples to the WindShift APT group.

Via the WhatsYourSign utility, we can confirm that indeed they are applications (not documents):



usrnode signed, but certificate has been revoked!

usrnode
/Users/patrick/Downloads/WindShift/Final_Presentation.app

```
item type: application
   hashes: view hashes
 entitled: none
sign auth: signed, but no signing authorities (adhoc?)
```

Moreover the utility indicates that the application (i.e. Final_Presentation.app) is neither fully signed and that its signing certificate has been revoked. We can confirm this with the codesign and spctl utilities:

```
$ codesign -dvvv Final_Presentation.app
Executable=Final_Presentation.app/Contents/MacOS/usrnode
Identifier=com.alis.tre
Format=app bundle with Mach-O thin (x86_64)
...
Authority=(unavailable)
Info.plist=not bound
TeamIdentifier=95RKE2AA8F
Sealed Resources version=2 rules=12 files=4
Internal requirements count=1 size=204

$ spctl --assess Final_Presentation.app
Final_Presentation.app: CSSMERR_TP_CERT_REVOKED
```

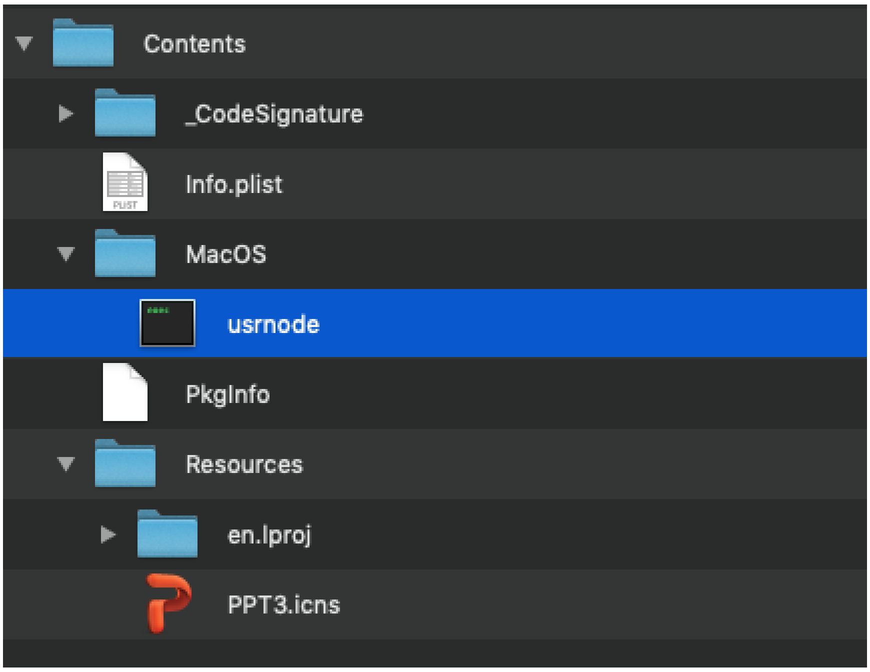
The fact that the signing certificate(s) of all the samples are revoked (CSSMERR_TP_CERT_REVOKED) means that Apple knows about about this certificate…and thus surely this malware as well. …yet the majority of the samples (3, of 4) are detected by *zero* anti-virus engines on VirusTotal.

Does this mean Apple isn't sharing valuable malware/threat-intel with AV-community, preventing the creation of widespread AV signatures that can protect end-users?! 🤔

Narrator: yes*

*of course sometimes they may not have permission (if the information was sourced from elsewhere).

Before diving into reversing/debugging these samples, let's take quick peak at their application bundles:



```
Contents
  _CodeSignature
  Info.plist
  MacOS
    usrnode
  PkgInfo
  Resources
    en.lproj
    PPT3.icns
```

First, note the main executable is named usrnode . This is also specified in the application's Info.plist file (CFBundleExecutable is set to 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 include `LSMinimumSystemVersion` which indicates the (malicious) application is compatible with OSX 10.7 (Lion), and `NSUIElement` key which tells the OS to execute the application without a dock icon nor menu (i.e. hidden).

However the most interesting key is the `CFBundleURLSchemes` (within the `CFBundleURLTypes`). This key holds an array of custom URL schemes that the application implements (here: `openurl2622007`). In the aforementioned blog post "Remote Mac Exploitation Via Custom URL Schemes", we described how WindShift (ab)used custom URL schemes to infect macOS systems…in exactly this manner! Yet another data point tying our samples to WindShift.

The `OSX.WindTail.A` sample described by Karim used a similarly named custom URL scheme: `openurl2622015`

Ok, let's dive in to look at some disassembly!

Loading the main binary `usrnode` into a disassembler (I used Hopper), we start at the `main()` function:

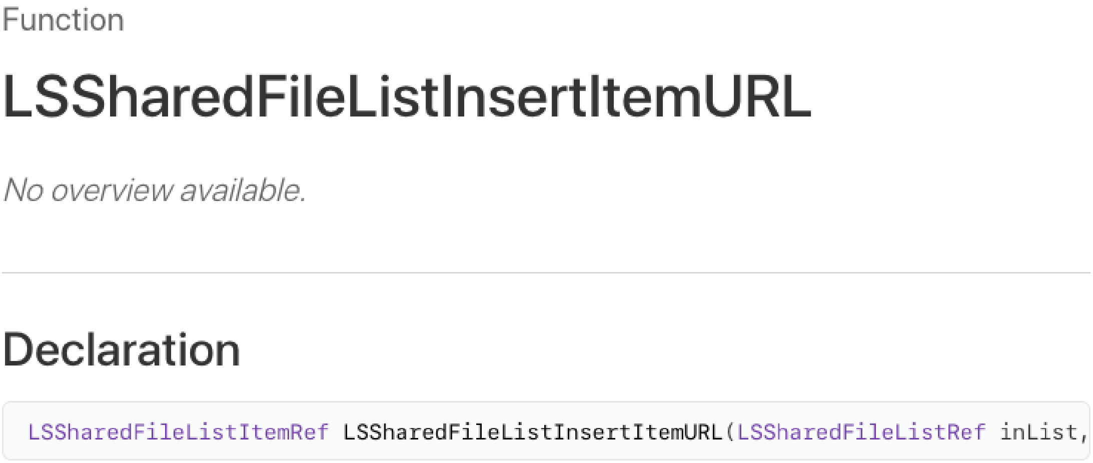
```
int main(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5) {

    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;
}
```

The `LSSharedFileListInsertItemURL` API is documented by Apple. Just kidding: “No overview available”:



Function

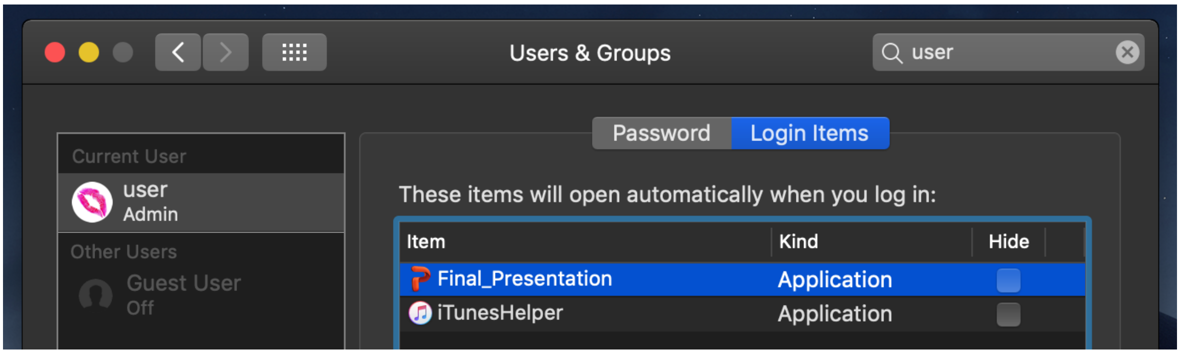
LSSharedFileListInsertItemURL

No overview available.

Declaration

```
LSSharedFileListItemRef LSSharedFileListInsertItemURL(LSSharedFileListRef inList,
```

So what does the `LSSharedFileListInsertItemURL` API do? It adds a login item, which is mechanism to gain persistence and ensure that the (malicious) application will be automatically (re)started everytime the user logs in. This is visible via `System Preferences` application:



These items will open automatically when you log in:

Item	Kind	Hide
Final_Presentation	Application	□
iTunesHelper	Application	□

…not the stealthiest persistence mechanism, but meh, gets the job done!

The `main()` function invokes the `NSApplicationMain` method, which in turn invokes the `applicationDidFinishLaunching` 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;
}
```

Pulling apart the above code, we can see: 1. The (malicious) application generates the current date/time, and formats it. 2. Builds a path to `date.txt` in it's application bundle (`Contents/Resources/date.txt`) 3. If this file doesn't exist, write out the (formatted) date and a random number 4. If this fails, set the `GenrateDeviceName` (sic) user default key to true 5. Read in the data from the `date.txt` file 6. invoke the `tuffel` method 7. Spawn a thread to execute the `mydel` method
Clearly steps 1-5 are executed to generate, then load, a unique identifier for the implant.

Let's observe this happening (via the `fs_usage` utility):

```
# fs_usage -w -filesystem | grep date.txt
00:38:09.784384  lstat64
/Users/user/Desktop/Final_Presentation.app/Contents/Resources/date.txt usrnode.8894
00:38:09.785711  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
```

The `tuffel` method is rather involved (and we'll expand upon in an update to this blog post). However, some of it's main actions include:

1. Moving the (malicious) application into the `/Users/user/Library/` directory
2. Executing this persisted copy, via the `open` command
3. Decrypting embedded strings that relate to file extensions of (likely) interest

We can observe step #2 (execution of the persisted copy) via my open-source process monitor library, ProcInfo:

```
procInfo[915:9229] process start:
pid: 917
path: /usr/bin/open
user: 501
args: (
    open,
    "-a",
    "/Users/user/Library/Final_Presentation.app"
)
```

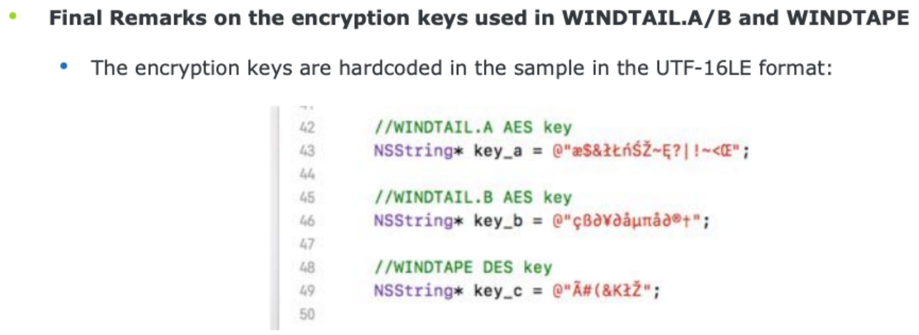
Step #3, (string decryption) is interesting as it both reveals the capabilities of the malware as well as (again) helps identify the (malicious) application as `OSX.WindTail`. The `yoop` method appears to be the string decryption routine:

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

Specifically it invokes a `decode` and `AESDecryptWithPassphrase` helper methods. Looking closer at the call to the `AESDecryptWithPassphrase` method, we can see it's invoked with a variable named `cfstring__` (at address `0x100013480`). This is the (hard-coded) AES decryption key:

```
cfstring___100013480:
  0x000000010001c1a8, 0x00000000000007d0,
  0x000000010000bc2a, 0x0000000000000010 ; u"æ$&łŁńŚŽ~Ę?|!~<Œ",
```

Interestingly this is the exact same key as Karin showed in his slides, for OSX.WindTail.A:



- **Final Remarks on the encryption keys used in WINDTAIL.A/B and WINDTAPE**
    - The encryption keys are hardcoded in the sample in the UTF-16LE format:

```
//WINDTAIL.A AES key
NSString* key_a = @"æ$&łŁńŚŽ~Ę?|!~<Œ";

//WINDTAIL.B AES key
NSString* key_b = @"çß∂¥∂åµπå∂®†";

//WINDTAPE DES key
NSString* key_c = @"Ã#(&KłŽ";
```

To see what the (malicious) application is decrypting, we can simply set a breakpoint within the `yoop` method, and then dump the (now) decrypted strings:

```
(lldb) b 0x000000010000229b
Breakpoint 8: where = usrnode`___lldb_unnamed_symbol6$$usrnode + 92, address =
0x000000010000229b
(lldb) po $rax
http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?very=%@&xnvk=%@
```

It's rather easy to break 'AES' when you have the key 🤣

Other strings that are decrypted (as noted) relate to file extensions of (likely) interest such as `doc` , `pdf` , `db` , etc. Makes sense that a cyber-espionage implant would be interested in such things, ya?

Moving on the `myDel` method appears to attempt to connect to the malware's C&C servers. Of course these are encrypted, but again, by dynamically debugging the malware, we can can simply wait until it invokes the AES decryption routine, then dump the (now) plaintext strings:

```
(lldb) x/s 0x0000000100350a40
0x100350a40: "string2me.com/qgHUDRZiYhOqQiN/kESklNvxsNZQcPl.php

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

The C&C domains (`string2me.com` and `flux2key.com`) are both WindShift domains, as noted by Karim in an interview with itWire

> "the domains string2me.com and flux2key.com identified as associated with these attacks"

These domains are currently 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
```

…thus the malware appears to remain rather inactive. That is to say, (in a debugger), it doesn't do much - as it's likely awaiting commands from the (offline) C&C servers.

However, a brief (static) triage of other methods found within the (malicious) application indicate it likely supports 'standard' backdoor capabilities such as file exfiltration and the (remote) execution of arbitrary commands. I'll keep digging and update this post with any new findings!

Conclusion

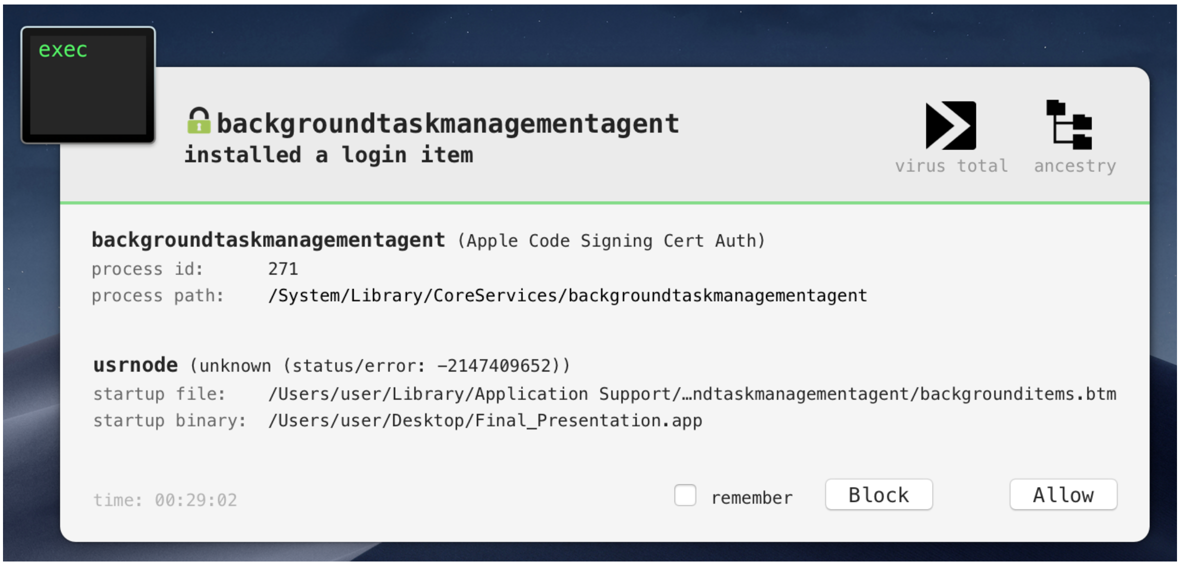
WindShift is an intriguing APT, selectively targeting individuals in the Middle East. Its macOS capabilities are rather unique and make for a rather interesting case study!

Today, for the first time, we publicly shared samples of a malicious application that I'm highly confident is `OSX.WindTail.A` (or is some variant thereof). This claim is based upon naming-schemes, unique infection mechanism, shared AES-decryption key, and some off-the-record insight.

In this blog post, we analyzed the OSX.WindTail to reveal its:

- initial infection vector
- method of persistence
- capabilities
- commmand & control servers

All that's left is to talk about detection an removal.

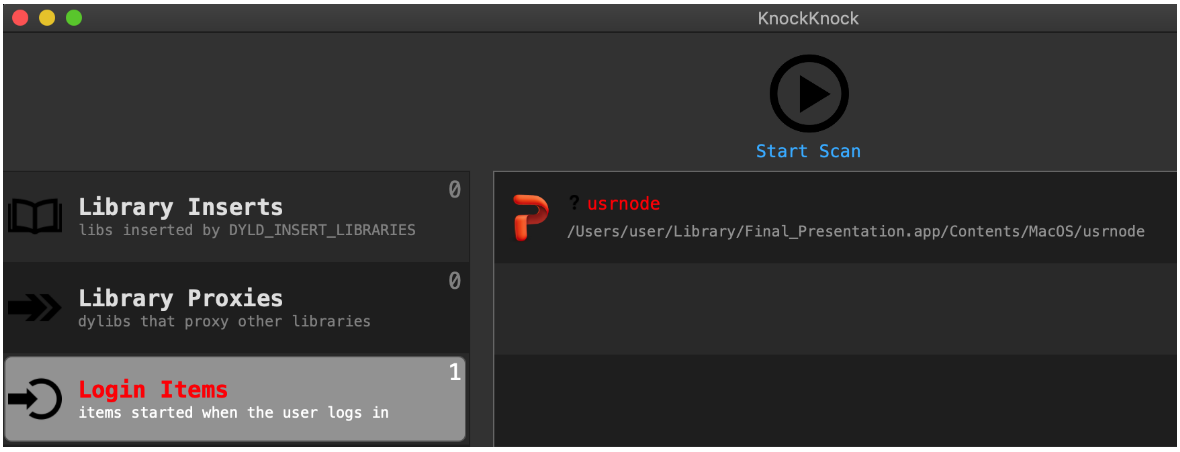


backgroundtaskmanagementagent
installed a login item

backgroundtaskmanagementagent (Apple Code Signing Cert Auth)
process id: 271
process path: /System/Library/CoreServices/backgroundtaskmanagementagent

usrnode (unknown (status/error: -2147409652))
startup file: /Users/user/Library/Application Support/…ndtaskmanagementagent/backgrounditems.btm
startup binary: /Users/user/Desktop/Final_Presentation.app

time: 00:29:02 □ remember Block Allow



Library Inserts 0
libs inserted by DYLD_INSERT_LIBRARIES

Library Proxies 0
dylibs that proxy other libraries

Login Items 1
items started when the user logs in

? usrnode
/Users/user/Library/Final_Presentation.app/Contents/MacOS/usrnode

…since current anti-virus engines (at least those found on VirusTotal) currently do not detect these threats, it's probably best to stick to tools (such as BlockBlock and KnockKnock) that can heuristically detect malware.

Though a tool such as KnockKnock is the suggested way to detect an infection, you can also manually check if you're infected. Check for a suspicious Login Item via the `System Preferences` application, and/or for the presence of suspicious application in your `~/Library/` folder (likely with a Microsoft Office icon, and perhaps an invalid code signature). Deleting any such applications and Login Item will remove the malware.

However if you were infected (which is very unlikely, unless you're a government official in a specific Middle Eastern country), it's best to fully wipe your system and re-install macOS!

Love these blog posts & tools? You can support them via my Patreon page!